Simulation Analysis of Data Sharing
in Shared Memory Multiprocessors

By

Susan Jane Eggers

B.A. (Connecticut College) 1965

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA at BERKELEY

Approved: Randy H. Katz ..... February 2, 1989
Chair Date
2/14/89
Feb. 7, 1989

****

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
FEB 1989		**00-00-1989 to 00-00-1989**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Simulation Analysis of Data Sharing in Shared Memory Multiprocessors	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This dissertation examines shared memory reference patterns in parallel programs that run on bus-based, shared memory multiprocessors. The study reveals two distinct modes of sharing behavior. In sequential sharing, a processor makes multiple, sequential writes to the words within a block, uninterrupted by accesses from other processors. Under fine-grain sharing, processors contend for these words, and the number of per-processor sequential writes is low. Whether a program exhibits sequential or fine-grain sharing affects several factors relating to multiprocessor performance: the accuracy of sharing models that predict cache coherency overhead, the cache miss ratio and bus utilization of parallel programs, and the choice of coherency protocol. An architecture-independent model of write sharing was developed, based on the inter-processor activity to write-shared data. The model was used to predict the relative coherency overhead of write-invalidate and write-broadcast protocols. Architecturally detailed simulations validated the model for write-broadcast. Successive refinements, incorporating architecture-dependent parameters, most importantly cache block size, produced acceptable predictions for write-invalidate. Block size was crucial for modeling write-invalidate, because the pattern of memory references within a block determines protocol performance. The cache and bus behavior of parallel programs running under write-invalidate protocols was evaluated over various block and cache sizes. The analysis determined the effect of shared memory accesses on cache miss ratio and bus utilization by focusing on the sharing component of these metrics. The studies show that parallel programs incur substantially higher miss ratios and bus utilization than comparable uniprocessor programs. The sharing component of the metrics proportionally increases with cache and block size, and for some cache configurations determines both their magnitude and trend. Again, the amount of overhead depends on the memory reference pattern to the shared data. Programs that exhibit sequential sharing perform better than those whose sharing is fine-grain. A cross-protocol comparison provided empirical evidence of the performance loss caused by increasing block size in write-invalidate protocols and cache size in write-broadcast. It then measured the extent to which read broadcast improved write-invalidate performance and competitive snooping helped write-broadcast. The results indicated that read-broadcast reduced the number of invalidation misses, but at a high cost in processor lockout from the cache. The surprising net effect was an increase in total execution cycles. Competitive snooping benefited only those programs that exhibited sequential sharing; both bus utilization and total execution time dropped moderately. For programs characterized by fine-grain sharing, competitive snooping degraded performance by causing a slight increase in these metrics.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**178**	

SIMULATION ANALYSIS OF DATA SHARING IN SHARED MEMORY MULTIPROCESSORS

by

Susan J. Eggers

Simulation Analysis of Data Sharing in Shared Memory Multiprocessors

Susan J. Eggers

Computer Science Division
University of California
Berkeley CA 94720

ABSTRACT

This dissertation examines shared memory reference patterns in parallel programs that run on bus-based, shared memory multiprocessors. The study reveals two distinct modes of sharing behavior. In *sequential sharing*, a processor makes multiple, sequential writes to the words within a block, uninterrupted by accesses from other processors. Under *fine-grain sharing*, processors contend for these words, and the number of per-processor sequential writes is low. Whether a program exhibits sequential or fine-grain sharing affects several factors relating to multiprocessor performance: the accuracy of sharing models that predict cache coherency overhead, the cache miss ratio and bus utilization of parallel programs, and the choice of coherency protocol.

An architecture-independent model of write sharing was developed, based on the inter-processor activity to write-shared data. The model was used to predict the relative coherency overhead of write-invalidate and write-broadcast protocols. Architecturally detailed simulations validated the model for write-broadcast. Successive refinements, incorporating architecture-dependent parameters, most importantly cache block size, produced acceptable predictions for write-invalidate. Block size was crucial for modeling write-invalidate, because the pattern of memory references within a block determines protocol performance.

The cache and bus behavior of parallel programs running under write-invalidate protocols was evaluated over various block and cache sizes. The analysis determined the effect of shared memory accesses on cache miss ratio and bus utilization by focusing on the sharing component of these metrics. The studies show that parallel programs incur substantially higher miss ratios and bus utilization than comparable uniprocessor programs. The sharing component of the metrics proportionally increases with cache and block size, and for some cache configurations determines both their magnitude and trend. Again, the amount of overhead depends on the memory reference pattern to the shared data. Programs that exhibit sequential sharing perform better than those whose sharing is fine-grain.

A cross-protocol comparison provided empirical evidence of the performance loss caused by increasing block size in write-invalidate protocols and cache size in write-broadcast. It then measured the extent to which read broadcast improved write-invalidate performance and competitive snooping helped write-broadcast. The results indicated that read-broadcast reduced the number of invalidation misses, but at a high cost in processor lockout from the cache. The surprising net-effect was an increase in total execution cycles. Competitive snooping benefited only those programs that exhibited sequential sharing; both bus utilization and total execution time dropped moderately. For programs characterized by fine-grain sharing, competitive snooping degraded performance by causing a slight increase in these metrics.

February 24, 1989

For my parents,
Jane Preston Morse Eggers
Harvey Heydorn Eggers

Acknowledgements

Graduate school takes a sizable chunk of one's life and is an experience much broader than researching and writing a dissertation. This acknowledgement is intended to thank those who helped me in a multitude of ways and shared those years with me.

I received good technical guidance from both faculty and students. My advisor, Randy Katz, took a chance on me when I looked like a dark horse. He gave me the opportunity to work on an exciting, multi-disciplinary project with extremely bright and capable students. At the same time he taught me that what I did with that opportunity was my responsibility. I am grateful for his counsel in matters that were technical, editorial and academic.

I would also like to thank the other members of my committee, Alan Smith and Ronald Wolff. Alan, in particular, provided insightful technical feedback that added greatly to the formulation of the sharing analysis and enhanced the clarity of the dissertation.

My colleagues on the SPUR project and the prelim study group provided comradeship and good technical conversations. Mark Hill, David Wood and Garth Gibson, in particular, were sounding boards for many of the results in Chapters 4 through 6, and Mark read and commented on the other chapters as well. Corinna Lee was my constant buddy in many areas; David Ditzel and George Taylor my anchors. These students and a few others were all members of "my dear lunch bunch", the crew with whom I spent so much of my graduate experience. It is memories of them that will make me tell my own students, while they roll their eyes in great disbelief (as I rolled mine, when Dave Patterson said this to me), that my graduate student days were the best in my life.

I would also like to thank those who provided the resources necessary for completing the empirical portions of the work. Andrea Casotto (CELL), Steve McGrogan (SPICE), Srinivas Devadas (TOPOPT) and Hi-Keung Tony Ma (VERIFY) donated the parallel programs and a considerable portion of their time discussing them. John Sanguinetti wrote the ELXSI tace generator; Dianne DeSousa helped with generating the ELXSI trace; Sequent provided the software on which the Sequent trace generator was based; and Frank Lacy generated and postprocessed the Sequent traces. Dominico Ferrari and Yale Patt provided resources for running many of the simulations. And Mark Manasse from Digital's Systems Research Laboratory served as consultant on competitive snooping issues in Chapter 6.

The research was supported by an IBM Predoctoral Fellowship, SPUR/DARPA contract No. N00039-85-C-0269, NSF grant No. 83-52227, Digital Equipment Corporation, and California MICRO (in conjunction with Texas Instruments, Xerox, Honeywell, and Philips/Signetics).

The last acknowledgement is rather unusual, but, on the other hand, so was I -- twenty years older than my fellow students, and a member of the minority sex. In the late nineteen sixties and early seventies I became involved in the resurgent women's movement. The experience taught me that women raised with the ideals of the fifties could still carve careers and live independent lives. Without that political message, the others in the movement, and the attitude they engendered, I would never have been here in the first place. I owe them a broadening of my life that was worth every sacrifice.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1. Motivation for this Dissertation

Shared memory multiprocessors are emerging as an important class of computer systems [Hill86, Olso85, Rose85, Thac88, Thak88]. One of the goals of this architecture is to improve performance by executing a single, parallel program on multiple processors. The programs share data and communicate with one another through a common main memory. The primary advantage of the shared memory is that it furnishes the programmer with the simplest parallel programming model, that of a single-level of globally accessible memory.[1] But because it is a single resource, used by all processors, it is a critical performance bottleneck.

The simplest shared memory architecture is one in which all processors and main memory are connected via a single system bus. (Figure 1-1 depicts a bus-based, shared memory mul-

[1] There are other advantages, of course. A major one is the ability to emulate other parallel processing memory organizations, such as message passing architectures.

tiprocessor.) This bus is the only communication path from the processors to memory, and among the processors. Therefore it is even a greater point of contention than main memory. The bus bandwidth determines how many processors the bus can support. Queueing delays in reaching memory via the bus can substantially increase memory access time and therefore program throughput.

Processor caches reduce the bandwidth demands on the system bus and shared memory [Good87b] by providing a distributed version of the single, shared memory resource. However, since multiple processors can now update different copies of the shared data, an additional problem of keeping all the versions consistent is introduced. *Cache coherency (consistency) proto-*

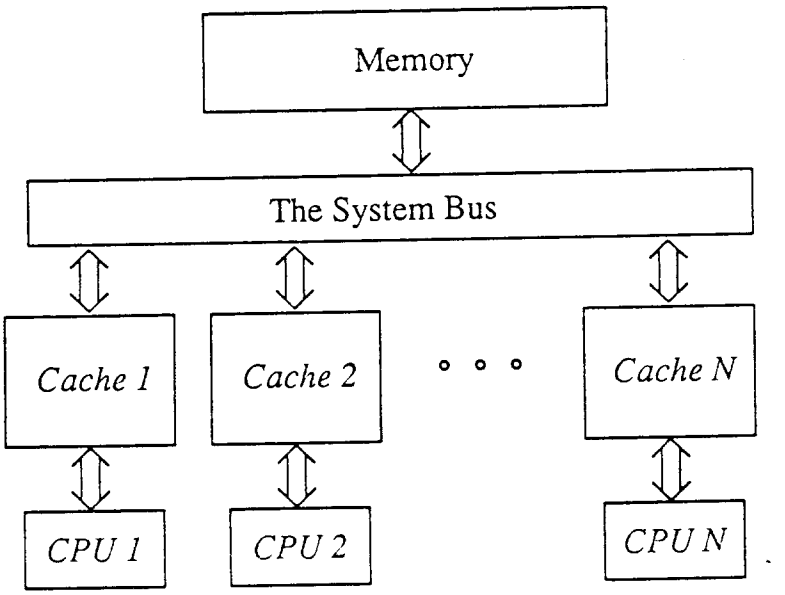

Figure 1-1: Bus-based, Shared Memory Multiprocessor

In a bus-based, shared memory multiprocessor all processor nodes and main memory are connected to a common system bus. This bus is the only hardware by which the processors may communicate with each other and with main memory. Therefore, despite the CPU caches, it is likely to cause the performance bottleneck.

cols describe operations for reading and writing shared memory that guarantee that a consistent view is maintained, i.e., a system with distributed caches behaves like one without them. Should multiple writes to a shared memory location occur simultaneously, it should be the case that (1) a value received on a memory read is the update of the last write to that location, and (2) the behavior of the coherency protocol is always predictable, i.e., no race conditions exist.

Numerous coherency protocols have been developed, both in hardware and software. Many were designed to provide good performance, usually meaning minimal additional bus traffic, under particular sharing conditions. For example, the goal of the write-broadcast protocols was to generate few additional bus operations when multiple processors were actively contending for the same shared addresses. Models of multiprocessor activity were also constructed to study the performance of several of the protocols.

All coherency protocols and multiprocessor models were developed in the absence of any real knowledge about the sharing behavior of parallel programs. Individual protocol optimizations were made, based solely on hypothetical assumptions of the sharing activity of these programs. The multiprocessor modeling was based on a workload model in which memory accesses to write-shared data were independent and uniformly distributed across all processors. If parallel program behavior deviates from the assumed behavior, then modeling results will mislead machine designers, and protocols will be adopted that produce less than optimal performance in actual machines.

The research in this dissertation takes the opposite approach. The *initial* goal of the work was to study the sharing behavior of parallel programs. Beginning with an analysis of actual program behavior has two benefits. First, it provides an understanding and characterization of the patterns of sharing *before* making a judgement about protocol and multiprocessor cache design. Second, the analysis of sharing is based on a real, rather than hypothetical workload, so

that the results are more representative of actual machine behavior.[2]

In some cases the results I shall present contradict previous conclusions about the merits of individual protocols and optimal cache organizations for multiprocessors. For example, conventional wisdom postulated that per-processor locality of reference for shared data was low [Arch86, Dubo82, Sega84, Vern86], and that any sharing in parallel programs would be characterized by inter-processor contention for shared blocks. Therefore, among the distributed, hardware protocols, it was predicted that write-broadcast protocols, which behave well under periods of contention, would have the least coherency-related bus traffic. And conversely, write-invalidate schemes would perform less well, presumably because of repeated misses on invalidated data. Results in this dissertation clearly demonstrate that there is a wider spectrum of sharing behavior in parallel programs than was assumed. Some programs do exhibit the hypothesized contention for shared addresses. However, others access blocks in a more sequential fashion, with one processor completing several accesses before another references the data. In other words, for many parallel programs there is good per-processor locality for shared data. For these programs, write-invalidate protocols are the better match; they generate both fewer cache misses and less bus traffic than write-broadcast.

Similar conclusions were made about the optimal cache block size for multiprocessor caches [Good83, Papa85, Sega84, Thac88, Vern86], the merits of read-broadcast protocols [Good87a, Sega84] and the insignificant effect of processor lockout from the cache on the overall performance of the snooping protocols. (The latter is inferred from the sole emphasis on minimizing bus traffic [Good83, Katz85, Papa85, Sega84, Thac88].) In all cases, results in this dissertation show that the inaccurate or simplified assumptions about the sharing behavior of the workload led to misleading or inaccurate results.

---

[2] As in all experimental research, the results are based on a finite amount of data, in this case traces of parallel programs. A different workload, for example, a larger number of programs, or a different application area, may produce dissimilar results.

Because of the presumption of poor locality of reference of shared data, it was postulated that large block sizes, a factor in obtaining good uniprocessor performance, would lower performance for parallel programs. My results show that for programs with sequential sharing, a large block size *improves* performance. In actuality, the optimal block size depends on the pattern of sharing within the cache block, in particular, whether it is sequential or fine-grain, rather than following a "the smaller, the better" rule.

Read-broadcast protocols allow caches that had previously invalidated data to refill their caches while the data is being bus transferred on another processor's read. The technique is considered to be a performance optimization, because it avoids all misses on invalidated data after the first. Results in this dissertation have found that, while read-broadcast does, in fact, reduce bus traffic, it can cause a loss in program throughput because it locks the processor from its cache. For some types of sharing behavior, protocols that increase processor lockout from the cache in order to reduce bus traffic can have worse overall performance than those that make the alternative tradeoff.

1.2. Organization of the Dissertation

The remainder of the dissertation is organized as follows. Chapter 2 contains a review of the coherency protocol literature. The review provides the reader with a summary of all coherency approaches, their advantages and drawbacks, and their distinctions from each other. While most of the protocols are not explicitly studied in the dissertation, the review provides a context for the development of those that are, namely, the write-invalidate and write-broadcast protocols.[3] Chapter 2 also contains a critique of published coherency protocol performance studies, based on analytic modeling and parameterized simulation.

[3] All terms will be defined and referenced where first discussed in detail.

Chapter 3 is a discussion of the methodology used in the dissertation research. All studies were done by trace-driven simulation of four parallel programs. Traditional trace-driven simulation techniques had to be extended in several ways to accommodate parallel processing requirements. For example, the traces had to be postprocessed to identify shared variables and inter-processor synchronization points. The resulting postprocessed traces were then too large for conventional disk storage; therefore a cache filter, specially designed for traces of parallel programs, was developed to compress them. Debugging the multiprocessor simulator was more difficult than developing uniprocessor simulators, because of the asynchronous nature of generating and satisfying global memory requests. Techniques were developed to trap system-wide debugging errors as they occurred dynamically, to prevent the incorrect actions of one processor from perturbing the behavior of others.

Chapter 4 develops a model of sharing that is used to determine the pattern of memory references to shared data, and the relative performance of write-invalidate and write-broadcast coherency protocols. There are three components to the development. The first characterizes those aspects of sharing that are important in measuring bus-related coherency overhead and defines metrics to reflect the characterization. Second, the characterization becomes the basis for an architecture-independent version of the model. Comparisons to simulation results verify this model's accuracy for the write-broadcast protocols. Finally, by progressively refining the model by incorporating specific cache parameters, most importantly, the size of the cache block, the model becomes a good predictor of coherency overhead for write-invalidate protocols as well.

The model development revealed two distinct modes of sharing behavior in the programs. In the first, *sequential sharing*, a particular processor makes multiple, sequential writes to the words within a block, uninterrupted by accesses from other processors. In the other, *fine-grain sharing*, processors contend for one or more words within the block and the number of per-processor sequential writes is very low. The results demonstrate that whether a program

exhibits sequential or fine-grain sharing affects the amount of coherency overhead incurred under a particular protocol, and with a particular block size.

Chapter 5 evaluates the cache and bus behavior of parallel programs under write-invalidate protocols over various block and cache sizes. The analysis determines the effect of shared memory accesses on both cache miss ratio and bus utilization by focusing on the sharing component of these metrics. The studies show that parallel programs incur substantially higher miss ratios and bus utilization than comparable uniprocessor programs. The sharing component of the metrics proportionally increases with both cache and block size, and for some cache configurations determines both their magnitude and trend. Again, the amount of overhead depends on the memory reference pattern to the shared data. Programs that exhibit sequential sharing perform better than those with fine-grain sharing. This suggests that writers of parallel software, in conjunction with better compiler technology, can improve program performance through better memory organization of shared data.

Both write-invalidate and write-broadcast protocols have been criticized for being unable to achieve good bus performance across all cache configurations. In particular, write-invalidate performance can suffer as block size increases; and large cache sizes will hurt write-broadcast. Read-broadcast and competitive snooping extensions to the protocols have been proposed to solve each problem. Chapter 6 provides empirical evidence of the performance loss caused by increasing the block size in write-invalidate protocols and the cache size in write-broadcast. It then measures the extent to which the solutions improve performance.

The results indicate that their benefits are limited. Read-broadcast reduces the number of invalidation misses, but at a high cost in processor lockout from its cache. The net effect can be an increase in total execution cycles. The competitive snooping protocol benefits only those programs whose memory reference pattern to shared data is one of sequential sharing. For programs characterized by inter-processor contention for shared addresses, competitive snooping

can degrade performance by causing a slight increase in bus utilization and total execution time.

The dissertation concludes in Chapter 7 with a summary of the research results, a discussion of the importance of the pattern of shared references in determining parallel program behavior and an outline of future research directions.

1.3. Research Contributions

Research contributions have come from both the research methodology and the results themselves. First, when the research was begun, there were no parallel traces available for analysis. The collection of multiprocessor traces that were generated for these studies have become one of two available to the research community. Second, analysis of the traces led to the development of a postprocessing methodology for synchronizing inter-processor memory references and a specialized cache filter for compressing traces of parallel programs.

The remaining contributions relate to the dissertation results themselves. A model of sharing, that incorporates per-processor locality of reference to shared data, was developed. Its purpose was to determine a program's pattern of sharing and the cost of maintaining coherent caches for write-broadcast and write-invalidate protocols. Although the model is quite simple, it was validated for both protocols via trace-driven simulation.

An analysis of cache memory design for multiprocessor caches was done over a wide range of cache and block sizes. The results pinpoint the additional cache misses and bus traffic incurred by parallel programs, and the cache configurations required to support varying levels of performance.

The analysis of cache coherency protocols revealed which design options produced the better performance and under what types of workloads and particular cache configurations.

Finally, and perhaps most importantly, the research highlighted the importance of sequential sharing behavior for minimizing coherency overhead and building accurate models of shar-

ing. The results were dramatic enough to warrant pursuing the development of programmer and/or compiler techniques to deliberately construct parallel programs that share sequentially.

1.4. References

[Arch86] J. Archibald and J. Baer, "An Evaluation of Cache Coherence Solutions in Shared-Bus Multiprocessors", *ACM Transactions on Computer Systems*, 4, 4 (November 1986), 273-298.

[Dubo82] M. Dubois and F. A. Briggs, "Effects of Cache Coherency in Multiprocessors", *IEEE Transactions on Computers*, C-31, 11 (November 1982), 1083-1099.

[Good83] J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic", *Proceedings of the 10th Annual International Symposium on Computer Architecture*, 11, 3 (June 1983), 124-131.

[Good87a] J. R. Goodman, "Coherency for Multiprocessor Virtual Address Caches", *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto CA (October 1987), 72-81.

[Good87b] J. R. Goodman, "Cache Memory Optimization to Reduce Processor/Memory Traffic", *Journal of VLSI and Computer Systems*, 2, 1 & 2 (1987), 61-86.

[Hill86] M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. Ousterhout and D. A. Patterson, "SPUR: A VLSI Multiprocessor Workstation", *IEEE Computer*, 19, 11 (November 1986), 8-22.

[Katz85] R. Katz, S. Eggers, D. Wood, C. L. Perkins and R. Sheldon, "Implementing a Cache Consistency Protocol", *Proceedings of the 12th Annual International Symposium on Computer Architecture*, 13, 3 (June 1985), 276-283.

[Olso85] R. Olson, "Parallel Processing in a Message-Based Operating System", *IEEE Software* (July 1985), 39-49.

[Papa85] M. S. Papamarcos and J. H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories", *Proceedings of the 11th Annual International Symposium on Computer Architecture*, 12, 3 (January 1985), 348-354.

[Rose85] C. D. Rose, "Encore Eyes Multiprocessor Market", *Electronics* (July 8, 1985), 118-119.

[Sega84] Z. Segall and L. Rudolph, "Dynamic Decentralized Cache Schemes for an MIMD Parallel Processor", *Proceedings of the 11th International Symposium on Computer Architecture*, 12, 3 (June 1984), 340-347.

[Thac88] C. P. Thacker, L. C. Stewart and E. H. Satterthwaite, Jr., "Firefly: A Multiprocessor Workstation", *IEEE Transactions on Computers*, 37, 8 (August 1988), 909-920.

[Thak88] S. Thakkar, P. Gifford and G. Fielland, "The Balance Multiprocessor System", *IEEE Micro* (February 1988), 57-69.

[Vern86] M. K. Vernon and M. A. Holliday, "Performance Analysis of Multiprocessor Cache Consistency Protocols Using Generalized Timed Petri Nets", *Proceedings of Performance '86 and ACM Sigmetrics 1986*, Raleigh NC (May 1986), 9-17 .

2 The Cache Coherency Protocols

2.1. Introduction

Cache coherency protocols have been implemented both in software and hardware. In the software protocols the programmer must identify all potentially shared objects, and for some protocols the coherency operations as well, so that the operating system or compiler can take appropriate steps (described below) to preserve consistency. For the simpler schemes the overhead of maintaining consistency is incurred whether or not sharing actually takes place during program execution. Recent advances in software protocols employ the automatic (compiler) detection of the read/write pattern of shared data to reduce this overhead.

Hardware solutions free the programmer and compiler from the responsibility of specifying coherency operations for the shared structures, and some incur the performance cost of preserving coherency only when the blocks are actively shared. These benefits occur at some cost in the hardware complexity of the cache and/or memory controller. The control of the

hardware protocols can either be centralized at the memory controller, using a global state directory, or be distributed among the caches and employ a snoop to track shared addresses.

A survey of all published protocols follows. Although only two classes of protocols, the write-invalidate and write-broadcast protocols, will be used in the studies in Chapters 4 through 6, I include a brief but comprehensive survey to provide a context for their development. The purpose of the survey is to (1) classify all coherency approaches and place particular protocols into the classification; (2) pinpoint the *main* advantages and drawbacks of the various coherency techniques; and (3) elucidate the most important distinctions between them. The reader is urged to consult the references at the end of this chapter for more details on particular protocols.

2.2. The Software Protocols

The programmer-controlled mechanisms used by the software protocols to enforce coherency are noncacheable pages, synchronization and cache flushing after critical sections. Declaring a page "noncacheable" forces all references to the page to access memory, thereby avoiding coherency problems altogether. The cost is an increase in bus traffic and slower execution, caused by the need to access memory on each shared data reference. Noncacheable pages are implemented by hardware that checks a cacheability bit in the appropriate page table entry during cache miss processing. If the bit is set, the block being referenced is passed directly to the CPU; if cleared, the block is cached. Noncacheable pages are used on the ELXSI 6400 [McGr86, Olso85], the Honeywell 60/66 [Saty80], the NYU Ultracomputer [Edlé85], IBM's RP3 (which has also implemented temporarily cacheable pages) [Bran85, Pfis85], the Intergraph Clipper [Neff86] and CMU's C.mmp [Full78].

An alternative approach requires a processor to access shared data via critical sections protected by semaphores. The processor first sets the lock, then references the data (caching it), and flushes the cache and releases the lock when finished. Normally the granularity of sharing is the page (segments are also used), and therefore only the shared pages need to be flushed,

rather than the entire cache. In the ELXSI 6400 semaphores are obtained and the buffer for the shared data is allocated through programmer-inserted operating systems calls. Since their accepted programming convention allows for reading shared data without locking it, the cache is flushed on lock as well as release.

Many systems, for example, the ELXSI 6400 and RP3, use a combination of the two mechanisms: noncacheable pages for storing the semaphores, and critical sections for the applications shared data. In RP3 the semaphores are counters, and may be maintained either by hardware or by busywaiting in software.

Early software protocols relied on programmer-specification of cacheable or noncacheable data. More recent schemes use the compiler both to automatically detect potential coherency violations and to prevent them by inserting either bypass-cache, or cache flush, invalidation or "post" instructions. The compiler separates a program into units, often delineated by loop boundaries, that are intended to be executed in parallel. The protocols differ in their treatment of shared data within each unit. In [Veid86] shared variables are cached, depending on the type of loop (doall is cached, doacross is not[1]); and, if cached, the entire cache is invalidated (called indiscriminate invalidation) after the unit has been executed, to prevent local reuse of the data. The coherency overhead is therefore a function of the loop type and bounds, rather than the number of processors or shared writes. In two other schemes the cacheable/noncacheable distinction is based on shared data usage. Shared variables are cacheable, if there are multiple readers and either no writing processors [Lee87] or only one writer (RP3 and the Ultracomputer [Bran85]). At the end of the unit's execution the [Lee87] scheme indiscriminately flushes the cache, allowing a copy-back memory update policy. The RP3 and Ultracomputer have a write-

[1] The type of loop is determined by the (data) dependence graph of the statements contained in it. Doall loops do not contain any cross-iteration data dependencies; therefore their iterations can be executed concurrently on multiple processors and their shared data can be cached. Doacross loops contain at least one cross-iteration dependency. This limits the concurrent execution of their iterations to a pipelined fashion, and shared data cannot be cached. The next iteration can be scheduled (on a different processor) after the the statements that contain the cross-iteration dependencies have been executed.

through policy; therefore their protocol can selectively invalidate (but in a single cycle) only the shared variables. (Write-through was used in RP3 and the Ultracomputer, because the burst of bus traffic caused by a cache flush caused more network delay in their multistage interconnection network than a steady, larger stream of write-through data [Edle85].)

The scheme proposed by Smith [Smit85] is a hardware optimization of the Ultracomputer protocol. In this protocol One Time Identifiers are associated with each cache block and with each page in the Translation Lookaside Buffer. A cache hit is determined by a comparison between the cache identifier and the cache address tag and the TLB identifier and the address of the memory reference. When the value of the TLB identifier is changed, the blocks on a particular page no longer hit, and the data becomes inaccessible. One Time Identifiers eliminate the need to invalidate the cache one block at a time, but have the disadvantages that (1) the granularity of shared objects is tied to the page and (2) reloading a replaced TLB entry will cause cache misses to its valid cache entries [Cheo88].

The three compiler-based protocols ([Bran85, Lee87, Veid86]) are conservative approaches; they specify invalidations or flushing for all shared data usage, whether it is dynamically required or not. Two other software protocols ([Cheo88, Cytr88]) were developed specifically to improve upon these earlier schemes by eliminating unnecessary coherency operations. They achieved this at the cost of an increase in the amount of compiler analysis required, and, for one, some hardware assist in the cache controller. [Cheo88] approximates constant-time, but selective invalidations, through reference marking[2] by source-specific (memory or cache) reads of shared data and invalidation bits associated with each cache block. A bit is set when the whole cache is invalidated (indiscriminately, in constant time), and cleared when its block is reloaded. The selectivity is accomplished by the dual reads. The cache-read instruction always reads from the cache (whether the bit is set or not, ignoring the invalidation); it is

[2] Reference marking tags each data reference as cacheable or noncacheable.

used when the compiler can guarantee that the cached data is current. Its memory-read counterpart reads from memory if the data is stale (the invalidation bit has been set) and otherwise from the cache; it is used when the compiler cannot tell the exact order in which references will be satisfied because of execution-time code scheduling on multiple processors. [Cytr88] obtains the same accuracy, but without hardware assists. In addition to the flush and invalidate instructions, their protocol uses a "post" instruction, which writes data to memory, but omits the invalidation. This allows the writing processor to continue referencing updated data in its cache, and any readers to obtain the most current value. The authors are concerned with applying their technique to automatically parallelized sequential programs. Their analysis takes advantage of the data and control dependency information generated by their parallelizing compiler (PTRAN [Alle88]) in order to generate as few coherency-related operations as possible. For example, the compiler only places invalidations in execution paths that contain a variable assignment, and only inserts posts after a processor's last assignment to a variable.

The advantages of the software schemes are that (1) unlike most distributed, hardware protocols, they don't require the broadcast capabilities of a shared bus, i.e., they are appropriate for more complex interconnection networks; (2) they avoid the runtime communication costs of the centralized hardware schemes that *can* operate on interconnection networks; and (3) they require little or no hardware support. However, they have several drawbacks. First, the simplest schemes place the burden of specifying the shared structures and the synchronization needed to handle them, and of debugging this code on the programmer. Secondly, software protocols require that sharing take place via memory, i.e., they preclude sharing through the cache-to-cache transfer mechanism of some of the distributed, hardware schemes (described below), which can reduce bus traffic. Lastly, and most importantly, the source level specification results in compiler generated (static) mechanisms for coherency enforcement. The compile time solution causes bus traffic to be generated whether it is required by the actual

Software Protocol Summary			
Protocol	Caching Policy	Flush Policy	Flush Determinant
ELXSI 6400	data type	selective	critical section
Clipper	never		
Honeywell 60/66	never		
NYU Ultracomputer	processor usage	selective	program unit
IBM RP3	data type, processor usage	selective	program unit
CMU C.mmp	never		
[Veid86]	loop type	indiscriminate	program unit
[Lee87]	processor usage	indiscriminate	program unit
One Time Identifiers	unspecified	selective	unspecified (presumably program unit)
[Cheo88]	data type	selective	data dependency analysis
[Cytr88]	data type (presumably)	selective	data/control dependency analysis

Table 2-1: Software Coherency Protocol Summary

This table contains a summary of the key differences among the software coherency protocols. The column, Protocol, contains either a coherency technique or the machine on which a particular coherency policy was implemented. Caching Policy specifies the criteria for allowing write-shared data to be cached. "Never" indicates that all write-shared data is noncacheable; "data type" that applications shared data is cacheable but locks are not; "processor usage" that cacheability depends on the number of readers and writers; and "loop type" that doall variables are cacheable but doacross are not. A "selective" Flush Policy flushes only a portion of the cache (the exact portion is dependent on the protocol); an "indiscriminate" one flushes the entire cache. Flush Determinant specifies when the flush is carried out. "Critical section" indicates on termination of a critical section; "program unit", on termination of some other compiler-determined unit of the program; "data dependency analysis" that flushing depends on the data dependency graph for the shared variables; "control dependency analysis" that it is determined by the execution path taken. The latter two approaches are also compiler-generated.

memory reference pattern to the shared data or not. The traffic appears in the form of memory accesses for noncacheable data, cache flushing after critical sections or program units and misses for data that have been needlessly flushed. (The latter is particularly expensive with the indiscriminate techniques.) Even the schemes that rely on the compiler detection of coherency violations suffer from this overhead. Within this group, the protocols that utilize reference marking, augmented by either hardware assists or data and control dependency analysis, should

achieve the best performance. It is still undemonstrated just how close to actual dynamic performance these protocols can come. (A classification summary of the software protocols appears in Table 2-1.)

2.3. The Centralized Hardware Protocols

The first hardware cache coherency protocols had a centralized controller that was responsible for maintaining consistency in all caches in the system. Cache transactions that could affect data coherency emanated from this central controller and were reported to it by individual caches. This included not only cache misses, but also those transactions that involved a change of cache state, even when there was no transfer of data. Each cache notified the central controller of any state change to a cached block, such as an update of a clean block. The controller signaled the change to all other interested caches, in this case a directive to invalidate their copy, and then transmitted permission to the requesting cache to modify the block. The central controller also initiated all copy-backs before a read of dirty (private) data.

Cache directories associated with each cache contained state bits for each block in the cache that determined whether there was a cache hit and whether a processor had permission to write to the block. In addition, the central controller maintained a global directory that was used in satisfying cache misses and enforcing coherency. The organization and content of the global directory depended on the particular coherency protocol. In the first of the centralized schemes developed, that of Tang [Tang76] (later implemented on the IBM 3081 [Gust82]), the cache directories were duplicated in the central directory. The drawback of this design was the global directory search required to locate all instances of a particular memory block. To eliminate the search time, other schemes maintained state that was associated with memory rather than cache blocks. The protocol developed by Censier and Fautrier [Cens78], the LSCS (Logical Semi-Critical Section) protocol of Yen and Fu [Yen82] and the protocol implemented in the S1 [Widd80] tagged each block in main memory with a presence bit for each processor in the

system (to indicate that the memory block was contained in a particular cache), and a modified bit (to indicate that a cached copy had been updated). These schemes eliminated the directory search of Tang's protocol, but tied the size of the global directory to the potentially cacheable blocks in main memory and limited the number of processors to the number of presence bits. Therefore they did not scale well with increasing memory sizes or numbers of processors. Three other protocols eliminated the latter drawback. The Two-Bit scheme of Archibald and Baer [Arch84] improved on the Censier and Fautrier design by storing only the cache state for each memory block. The number of bits per memory block was thus independent of the number of processors in the system. This variation reduced the amount of memory devoted to the global directory and allowed the number of processors to be expandable. Its drawback was that it was not known globally which cache held which block, and therefore all caches had to be polled to see which should invalidate on a write. Dir_1NB and Dir_1B[3] [Agar88b] distribute the global directory and its operations among the caches. Both schemes limit the presence bits, and therefore the caches that can contain a block, to one. (The "presense bit" is actually a pointer to the appropriate cache.) They will perform well only if data is shared in a very sequential nature, with one processor accessing it at a time. Dir_1B is an optimization of Dir_1NB; it provides one additional bit to indicate that there are multiple shared copies. When this bit is set, the coherency operation is broadcast, rather than sent point to point. (A summary of the centralized protocols appears in Table 2-2.)

Like the software protocols, the centralized schemes are well suited for complex interconnection networks, i.e., multiple paths to memory. In addition, their sequential operation is easier to design and debug. However, several disadvantages make them unsuitable for multiprocessors that utilize a single path (and therefore broadcast-based) interconnect, such as a bus. The most serious drawback is their adverse effect on bus utilization, caused by the need

[3] Their notation indicates that only 1 cached copy is allowed, and that bus broadcasts cannot (NB) or can (B) be used to determine which cache contains the data.

Centralized Coherency Protocol Summary	
Central Directory Organization	Protocol
State of all caches	Tang
State per memory block Presence bit per processor	Censier & Fautrier LSCS S1
Processor id per memory block Processor id + broadcast bit per memory block	Dir_1NB Dir_1B

Table 2-2: Centralized Coherency Protocol Summary

This table contains a summary of the key features of the centalized coherency protocols.

for separate communication between the central controller and each cache. This overhead is directly proportional to the number of caches that share the data. Second, additional overhead also results from the memory update of cached dirty data before it is shared. Third, the global directory may need to be changed even when the action involves a single cache, for example, signaling the central controller when a clean block is replaced. Other disadvantages are the inability to allow for processor expansion (with the exception of the Two-Bit, Dir_1NB and Dir_1B schemes), the extra memory needed for the global directory and the time consumed by searching it (Tang).

2.4. The Distributed Hardware Protocols

2.4.1. Overview

When the interconnection between processors and memory is a single bus, the generality of the centralized coherency protocols may no longer be necessary. With multiple-cache data sharing, the one-to-one communication between the global and cache directories generates an amount of bus traffic that would quickly consume all available bus bandwidth. On the other hand, the broadcast capability of the shared bus provides simultaneous transmission of informa-

tion to all processors. This feature led to a distributed approach to cache coherency. Under distributed coherency protocols, the responsibility for maintaining consistent caches belongs to the individual cache controllers rather than to a central controller; and the need to maintain a global directory is eliminated. Consequently, the number of processors can be extended to the limit of the bus bandwidth.

In distributed coherency protocols a portion of each cache controller, the *snoop*, continuously monitors the system bus for operations taking place on blocks contained in its cache. When a match is made between the address of the bus operation and one of the cache tags, the snoop performs a consistency-preserving operation, based on the type of bus request, the state of the cache block and, of course, the particular protocol. For example, if the bus request is a write and the cache block state indicates that the block is shared, for several of the distributed protocols, the snoop will invalidate its cache entry. Within the distributed, hardware category, all protocols follow one of two approaches to maintaining coherency: write-invalidate or write-broadcast. (Again, a summary for the distributed, hardware protocols appears in Table 2-3.)

2.4.2. The Write-Invalidate Protocols

Write-invalidate protocols maintain coherency by requiring a writing processor to invalidate all other cached copies of the data before updating its own. It can then perform the current update, and subsequent updates (provided there are no intervening accesses by other processors) without either violating coherency or further utilizing the bus. The invalidation is accomplished by an invalidating bus operation. Caches of other processors monitor the bus via the snoop portion of their cache controllers. When they detect an address match, they invalidate the entire cache block containing the address. Because they create a data writer that can access a shared block without using the bus, write-invalidate protocols should minimize the overhead of maintaining cache coherency in two cases: when there are multiple consecutive writes to a block by a single processor, and when there is little contention for the shared data.

Distributed Coherency Protocol Summary			
Protocol	Category	Memory Update Policy	Unique Feature
Write-Through with Invalidation	WI	Write-through	
Write Once	WI	Copyback	Reserved state
Synapse N+1	WI	Copyback	Explicit memory ownership
Berkeley Ownership	WI	Copyback	Owned Shared state
Illinois	WI	Copyback	Private Clean state
RWB	WI	Copyback	Read Broadcast
Bitar	WI	Copyback	Got-Lock & Need-Lock state
VMP	WI	Copyback	Software implementation (interrupt-driven) Unlocked bus operation
Firefly	WB	Copyback for private, Write-Through for shared data	Memory updated with broadcast
Dragon	WB	Copyback for private, Write-Through for shared data	Memory not updated with broadcast
Competitive Snooping	WB & WI	Copyback for private, Write-Through for shared data, Copyback after the invalidation	Switches coherency policy
Clipper	WB	Depends on cache state	Snoops on shared bus operations only

Table 2-3: Summary of the Distributed, Hardware Coherency Protocols

This table contains a summary of the key features of the distributed, hardware coherency protocols. WI indicates that the protocol is one of the write-invalidate protocols; WB that is write-broadcast.

The simplest write-invalidate protocol is Write-Through with Invalidation, which has been implemented on dual processor machines[4] (IBM 370/168 [Saty80], IBM 3033 [Smit85] and the VAX 11/780 [Arch84]) and more recently shared memory multiprocessors (the Sequent Balance 8000 [Thak88] and Encore Multimax [Bell85]). Under this protocol each write to the

[4] One processor was dedicated to I/O operations.

cache is propagated through to main memory.[5] As usual, other cache controllers snoop on the addresses of the writes and invalidate their copies if there is a match with their own cache tags. The addresses are broadcast on a special high speed bus on the IBM machines, while the system bus is used in the Sequent and Encore. The major drawback of Write-Through with Invalidation is the amount of bus traffic generated by the writes. It forces the coherency-related bus traffic (as well as bus traffic for private data) to be a direct function of the number of writes, rather than of the amount of sharing.

The other write-invalidate protocols follow a copy-back policy for updating memory. At a minimum, these protocols use the normal read and write bus operations and three state values (invalid, read only and possibly shared, and exclusively held and therefore writable) to guarantee consistency in the caches [Arch84]. For reasons of bus efficiency, most introduce a unique fourth state and some have special bus operations. These additional features are used to improve bus utilization when detecting and handling shared data. For example, an invalidation signal is used by a writing processor to invalidate other cached copies of the block being updated; the clean, private state is used to eliminate the need for issuing this invalidation signal on the first write to an exclusively held block.

The first of the copy-back, write-invalidate protocols to appear in the literature was Write Once [Good83]. (Since there is only one write-through protocol, Write-Through with Invalidation, from now on the term, "write-invalidate protocol", will refer to the copy-back subset.) Write Once employs write-through on the first write to a block (during which all snoops invalidate their copies, and the updated block's state is changed to Reserved) and copy-back on all subsequent writes (until a read by another processor). It provides for cache-to-cache transfers for requests for cached dirty data, but requires a subsequent memory update to cleanse the

[5] Write-Through with Invalidation is unlike other write-invalidate protocols, because its write-through memory update policy precludes taking advantage of private copies of shared data on cache updates. I am including it in the write-invalidate category, because it utilizes invalidation signals to maintain coherency.

cached copies.[6] Thus it incurs additional bus traffic over other protocols in two situations: over those protocols that never rely on write-through for multiple writes to cached blocks (by a single processor) [Katz85] and over those that either update memory during a cache-to-cache transfer or only on block replacement for shared writable data.

The Synapse N+1 developed a different protocol, based on the concept of block ownership [Fran84a, Fran84b]. Both memory and snoops could be explicit owners of blocks, and therefore directories were associated with each. Ownership by a cache (i.e., private ownership) carried the right to update the block locally without initiating a bus transfer and the obligations both to update main memory on block replacement and to provide data to other caches upon request. Obtaining private ownership involved a bus transaction that caused other caches to invalidate their copies of the block. The protocol avoids the extra bus operation to memory incurred by Write Once for write requests that result in cache-to-cache transfers of dirty data, but pays a stiff three transaction penalty for reads (the bus operation is aborted, main memory is updated, and the bus operation is then retried).

The Berkeley Ownership protocol [Borr85, Katz85], developed for the SPUR multiprocessor [Hill86], improved upon the Synapse scheme by eliminating both the three bus transaction overhead on reads that were satisfied by a cache and the state directory associated with memory. All transfers between caches are done in one bus transfer, and memory is not updated in the process. The notion of shared, but still owned and possibly dirty, data is preserved by the introduction of the Owned Shared state. If no cache owns a block, then memory is considered the implicit owner, thus eliminating the need to explicitly represent memory ownership with additional state. (See Chapter 4, section 2 for a more detailed description.)

The Illinois protocol [Papa85] introduced the clean, private state to distributed protocols. The use of this state eliminates the need to signal a bus invalidation when it is known that the

[6] Memory must be updated because there is no cache state value denoting "shared, dirty" data.

processor has the only cached copy of the block.[7] It thus reduces bus traffic to the number of cache misses, invalidations when the data is thought to be shared and writes to memory on block replacement. The clean, private state was used in lieu of the owned states. The lack of a block owner means that any snoop that has a copy of a particular block might respond to a read request. This requires either (1) an implementation that will guarantee that all snoops can respond in the same bus cycle, or (2) extra processing time and/or logic to arbitrate for the multiple snoop/memory responses and to retract the bus requests for the losers. In the implementation proposed for this protocol, memory was updated during a processor request for dirty data. The simultaneous update reduces the number of bus transfers for block replacement below that incurred in Write Once, Synapse and Berkeley Ownership, but requires a custom-designed snooping memory controller to prevent memory latency from dominating the time of the data transfer. (An almost identical protocol has been implemented on the Sequent Symmetry [Love88]. The differences between them are that in the Symmetry protocol a cache with modified, exclusive data changes its state to invalid, rather than shared, and there are no cache-to-cache transfers of shared, clean data.)

Rudolph and Segal [Sega84] and Bitar and Despain [Bita86] have designed protocols which are intended to optimize synchronization. The former, called RWB, is based on a read-broadcast mechanism in which snoops take data from the bus (on data transfers initiated by some other processor) if their cache blocks for the data are currently invalid. The write policy is write-broadcast (explained below) for the first write to a block and an invalidation signal for the second. It is not clear how this broadcast-invalidation sequence benefits semaphore usage, since semaphores are written twice per critical section (once for setting, again for clearing). The one write-broadcast precludes a private write for lock clearing (which would occur had the

[7] The version of Berkeley Ownership that was implemented on SPUR approximates the private, clean state by including an addressing mechanism for detecting references to the stack. Separate invalidation signals are not issued for these references..

first write triggered an invalidation instead), and the invalidation signal nullifies the data in other caches, just when they need to detect that the lock is available. A better invalidation point would be after two write-broadcasts. This would allow locks to remain in all caches, always with the most current value, but other shared data to be invalidated after two writes. The scheme seems better designed for the multiple readers/single writer situation. Other drawbacks of the protocol are that (1) because of the write-broadcast it is intrinsically tied to a one-word block size (unless the cache block state includes a valid bit per word rather than a single state value for the entire cache block) and (2) the additional snoop accesses to the cache for the read-broadcasts can interfere with CPU processing (see Chapter 6).

The Bitar and Despain protocol introduced two special coherency states to be used in lieu of explicitly setting and clearing locks. One signifies that a cache has locked a data block; the other that other processors are waiting for the locked block. If the block remains in the cache until the unlock, the additional states reduce the bus operations needed for locking and unlocking. However, if the block is replaced, its locked state will be lost. Therefore a technique for storing and checking the locked state in the block must be adopted, in addition to the extra cache states. Their proposed implementation for busywaiting requires a special bus arbitration scheme to give maximum priority to waiting processors, a special busy-wait register that contains the address of the lock (for waiting processors) and a snoop for the busy-wait register that monitors the bus for the unlock, obtains the lock and interrupts the processor to begin executing the critical section. The obvious drawback of this implementation is the complexity of the additional hardware. The advantage is that the busy-wait register and its snoop eliminate rereading the semaphore after it has been unlocked. This penalty is paid in the other write-invalidate protocols.

The protocol implemented in the VMP multiprocessor [Cher86, Cher88] is a hybrid between the write-invalidate and software coherency schemes. Like the write-invalidate protocols, it utilizes a snoop (which they call a "bus monitor") to monitor the backplane for

coherency bus operations. However, the snoop's actions are implemented in software, rather than as part of the cache controller hardware. The snoop does its lookup on an action table (rather than the cache tags), that contains a bus operation-dependent action for each cache block-sized unit of main memory (called cache page frames). For each action that requires a snoop response, the CPU is interrupted; it then executes interrupt handler code from local memory. The coherency protocol that is implemented is ownership-based (very similar to the Synapse protocol, including the three bus operation sequence for reading dirty data), with an additional feature: a special bus operation that can be used to signal that a lock has been unlocked. The CPU delay to perform all operations caused by the interrupt-driven coherency implementation is an extra source of overhead for VMP, relative to the other write-invalidate protocols. Its attraction is the ease with which the software algorithms can be debugged. Furthermore, VMP's operating system includes a routine for differentiating between shared and private data, which results in optimizations identical to those produced by the private, clean state of the Illinois protocol and the stack segment of the Berkeley Ownership implementation on SPUR.

4.3. The Write-Broadcast Protocols

Rather than invalidating cached copies of shared, writable blocks, write-broadcast protocols broadcast writes to shared blocks, so that all caches and memory have access to the most current value. Blocks are known to be shared through the use of a special bus line. Snoops assert this signal whenever they address match on an operation for a block that resides in their caches. As long as a writing processor detects an active shared line, it will continue to issue the broadcasts. In the absence of an active shared signal, the processor will complete the write locally. Thus, the signal provides for write-through for shared data, but allows a copy-back memory update policy to be used for private data.

Write-broadcast protocols have potential performance benefits for both private and actively shared blocks. First, an inactive shared line prevents needless bus operations to data that reside only in the cache of the writing processor. In addition, because it broadcasts all shared updates, write-broadcast avoids the pingponging of shared data among the caches that would occur with the invalidations of the write-invalidate protocols during periods of data contention. However, for data that is shared in a sequential fashion, with each processor completing all its accesses to the data before another processor begins, the write-through policy for shared data may degrade bus performance.

Write-broadcast protocols were proposed for the Xerox PARC Dragon [McCr84] and have been implemented on the DEC Firefly [Thac88]. The difference between the two protocols is that the Dragon updates memory only on block replacement (in a procedure identical to the Berkeley Ownership protocol), while the Firefly updates simultaneously with each write to shared data. (See Chapter 4, section 2 for a more detailed description of the Firefly protocol.)

Competitive snooping [Karl86, Karl88] is a write-broadcast protocol that switches to write-invalidate when the breakeven point in bus-related coherency overhead between the two protocols is reached. This point occurs when the number of cycles for the broadcasts issued equals the sum of the cycles should all processors need to reread the invalidated data. Their proposed implementation assigns a counter, whose initial value is the cost in cycles of a data transfer, to each cache block in every cache. With each snooped broadcast, some cache's counter is chosen to be decremented. When its value is zero, the block is invalidated. When all counters are zero, the write-broadcasts cease. Any access by a processor resets its cache's counter to the initial value. Competitive snooping limits the overhead of write-broadcast to twice that of optimal, but at some cost in hardware complexity. Finite state machines for write-invalidation and write-broadcast protocols, bus lines for picking the counter to be decremented and state bits (per cache line per cache) for counting must be implemented. (A more detailed description of two competitive algorithms appears in Chapter 6.)

One of the Clipper's coherency mechanisms is a combination of the software and write-broadcast approaches. The compiler designates all cache lines as shared or private, copy-back or write-through. When the cache state indicates shared/write-through, the snoops monitor the bus write, and update their caches if there is an address match. The technique reduces the number of snoop lookups on the single-ported cache state RAMs. (Private data is not snooped.)

2.4.4. IEEE Classification of Distributed, Hardware Protocols

Sweazy and Smith [Swea86] devise a general classification for all existing distributed, hardware coherency protocols. The model includes the union of the states of all the distributed protocols (shared unmodified, shared modified, exclusive modified, exclusive unmodified and invalid); and their standard backplane implementation (the Futurebus) includes signals to implement all coherency operations. The resulting protocol is therefore a superset of the distributed protocols and would allow caches with different protocols and memory update policies to communicate successfully.

2.5. Initial Performance Studies

The early research in cache coherency focused on algorithmic development (the protocols), proofs of functional completeness by case analysis[8] (using Markov chains of the coherency states and the bus operations that cause transitions between them, e.g., [Sega84, Yen85]), and a static analysis of the performance of individual bus operations. More recent work has focused on a dynamic analysis, via analytic modeling[9], parameterized simula-

---

[8] Case analysis is a method of verifying the functional completeness of an algorithm by an exhaustive examination of the ramifications of each input condition.

[9] An analytic model is a mathematical description of a system that solves for steady state behavior. It is a static description, expressing the dependencies among the system parameters. The parameters in the model are random variables.

[10] A parameterized simulation is a description of system behavior over the time domain. Its inputs are synthetic events whose values are drawn from probability distributions.

tion[10] and trace-driven simulation.[11] In the analytic work, simplifying assumptions were often made to reduce the computation time needed to solve the models. For example, several studies simplified the complex interactions between the caches and the bus and omitted entirely the interference between the snoop and the CPU's side of the cache controller. The most serious simplification was for workload behavior.

Patel [Pate82] modeled processor utilization (the amount of time it took a processor to accomplish one unit of work) of a multiprocessor system with a write-through cache. In the absence of any knowledge of the nature of sharing, his model assumed that multiprocessor memory references were random, independent and uniformly distributed over all of memory, i.e., there was very little sharing of any kind. Therefore the activity of one processor could be modeled and then multiplied by the number of processors. His study found that processor utilization is a function of miss rate times the data transfer time.

Dubois and Briggs [Dubo82] modeled a shared memory multiprocessor with a centralized coherency scheme. Their model more accurately describes the details of coherency activity, and also includes separate workload models for write-shared data and the other types of memory accesses. To emulate the more complex coherency mechanisms, they modeled the effect of transmitting the invalidation signals and of a processor's waiting for the release of read/write data. However, their model still omits the effects of contention for the global directories, which biases their results optimistically. The reference stream for their simulation was a merging of private and shared read-only accesses, generated using the LRU stack model[12], and

---

[11] Trace-driven simulation uses a trace reflecting the execution behavior of the program under study as its input. For coherency studies the trace is composed of memory references issued by all processors in the multiprocessor.

[12] The LRU (Least Recently Used) stack model models a reference string (a series of memory addresses) with an LRU distance string. It is one of the priority stack techniques. Priority stack techniques are a method of simulating multiple-sized caches concurrently. They assume that (1) larger caches always contain the blocks that are resident in the smaller caches, (2) the last referenced cache block is on the top of the stack and is the only block on the stack to move up the stack, and (3) no blocks below the old position of the referenced block move. The LRU version uses the least recently used block as the victim for stack replacement [Coff73].

shared read-write references, based on an independent reference model[13]. (As in the Patel model, the latter is used because it is assumed that shared data has poor locality of reference.) The analysis assumed very small levels of sharing (a large amount was considered to be one percent), and all simulations were done with approximately 1000 addresses. Nevertheless, their results support intuitive notions about the sources of performance degradation in centralized coherency schemes. They found that coherency invalidations increased the miss ratio, the traffic required to enforce the coherency rules (primarily copy-backs associated with the invalidations) and access the global directory, and the amount of time the processor was blocked from the cache. All metrics increased proportionally to the degree of sharing. The degradation due to processor lockout from the cache for a state change (valid to invalid, private to shared) was found to be insignificant.

The Dubois-Briggs workload model was important, because it served as the basis for three studies of distributed protocols [Arch86, Vern86, Vern88]. [Vern86] modeled particular features of coherency protocols, rather than the protocols themselves, using generalized timed Petri Nets[14]. They evaluated the effects of adding each of the features to a base coherency protocol very similar to Write Once. The four features were a shared bus line that could be used to implement a private clean state,[15] the Owned Shared state of the Berkeley Ownership protocol which allows cache-to-cache transfers of dirty data without the need to update memory, an invalidation signal and the write-broadcast mechanism, assuming use of the shared line. They measured bus utilization and processing power[16] for data caches only. Their results indicated that the shared line provided the biggest performance advantage, particularly as the level of

[13] In the independent reference model the probability of a reference to a particular next block is fixed; it depends neither on the block previously referenced nor on whether the current block was referenced before. In other words, the model does not reflect locality of reference.

[14] Generalized timed Petri Nets is an analytic technique whose state transitions have a deterministic firing duration, but the next state is stipulated by a probability distribution.

[15] See Section 2.3.3 for a description of the use of the line.

[16] Processing power is the number of processors times their average CPU utilization.

sharing was increased. They also found that the invalidation signal and dynamically switching from write-broadcast to write-invalidate (as in the protocols described in [Sega84] and [Karl86, Karl88]) contributed a negligible performance improvement over Write-once. It should be noted, however, that their model's probabilistic inputs dictate a homogeneous access pattern to the shared data, across processors. This reference pattern, one in which all processors have equal access to the shared data, i.e., there is poor per-processor locality of reference, is one in which write-broadcast *should* perform well. Conversely, write-invalidate should behave relatively poorly, because the write-shared blocks will ping-pong among the caches. Therefore their choice of workload model biased the results; and the poor performance of the write-invalidate protocols should come to no surprise. They also found that a one-word block size performs best, and for a similar reason. Here the uniform accesses to shared data were coupled with a pessimistic hit ratio (.5) for write-shared data.

Identical studies are performed in [Vern88], but using a simpler modeling technique, Mean-Value Analysis,[17] in place of the generalized timed Petri Nets. Their studies indicate that the simpler methodology yields results comparable (within 3 percent on average) to the more detailed model, across all protocol features and a wide range of parameter values. (There is a slight tendency to underestimate bus utilization and overestimate inter-processor memory and cache interference, but the differences are all tolerable.) The paper convincingly demonstrates the superiority of the simpler approach, in terms of both the accuracy of its results, and its efficiency in obtaining solutions, and therefore its ability to solve for larger systems. However, because Mean Value Analysis suffers from the same probabilistic treatment of the shared data reference input stream as generalized timed Petri Nets, again, the results are biased toward write-broadcast.

[17] Mean-Value Analysis is an analytic technique that solves a set of equations that compute the mean value of certain performance (output) metrics in terms of the mean value of the model's inputs.

Archibald and Baer [Arch86] compared the behavior of several of the distributed, hardware protocols via parameterized simulations. Unlike previous comparative protocol studies, they tried to make the specification of a protocol independent of its implementation. For example, they disallowed the simultaneous updating of memory during a block transfer to a cache that is used in the Illinois protocol. Again, their workload model was derived from the one developed by Dubois and Briggs, and their metric for protocol comparisons was one similar to processing power. Their analysis focussed on scenarios of high and low (i.e., mostly private data) contention for shared blocks. Many of their results are intuitive. They found that the performance for all protocols was comparable for a small number of processors; but that as the number of processors increased, protocols that could detect private, clean data (i.e., the write-broadcast and Illinois protocols) performed best, assuming a low amount of sharing; with more contention for shared data (modeled by shared block references of five percent on sixteen shared blocks) the write-broadcast protocols did best. One counter-intuitive result is that with higher levels of sharing, Berkeley Ownership performed better than the Illinois protocol. While it is true that the Berkeley protocol has a more efficient handling of shared data (because of the Owned Shared state), the latter is better at handling private data (because of the clean private state). Since private data is referenced (in their study) the vast majority of the time (95 percent), Berkeley Ownership's better performance over the Illinois protocol protocol is surprising.[18]

[Owic89] modeled the two basic software coherency mechanisms, noncacheable shared data and cache flushing, with both a bus and cross-bar interconnect. The model includes a component for bus and interconnect contention, as well as for the workload and system operations. The results support intuitive notions about the performance of these protocols. At low levels of

[18] In this study the cache sizes were small (2K and 16K bytes). A larger cache would have exaggerated the overhead of sharing, which this research has shown to be relatively immune to the benefits of increasing cache size. See Chapter 5.

sharing with few processors there is little difference in protocol performance. As these parameters rise, the effectiveness of noncacheable data falls off rapidly. (At high levels, it saturates the bus with a processing power less than two.) A sensitivity analysis of several model parameters in the bus studies indicates that cache flushing is quite sensitive to the number of data references, particularly shared data references, and the number of accesses to a block before it is flushed. A comparison of the software techniques to a representative distributed, hardware protocol (Dragon) indicates that the snooping approach gets better performance at all parameter values. In the multistage interconnection studies, since network bandwidth increases with the number of processors, both software protocols scale well. Unlike the previous analytic studies, this work includes a workload parameter that reflects per processor locality of reference to write-shared data. The parameter measures the number of accesses to a write-shared variable before it is flushed from the cache and is used to approximate the number of writes before flushing.

The remainder of the work on coherency protocol performance has a more direct bearing on the topics covered in this dissertation and therefore will be discussed in conjunction with the results of particular studies (see Chapters 4 through 6). All experiments were done using trace-driven simulation. The traces were generated in 400K reference snapshots on a four-processor VAX 8350 running the MACH operating system [Baro85, Site88]. Since the generation technique was microcode-based, the traces contain references from a cross-context workload, including the operating system. Briefly, [Site88] analyzed several aspects of multiprocess and parallel processing cache behavior, the most relevant of which for this work was the measurement of the additional bus traffic caused by the invalidations in write-invalidate protocols. [Agar88a] studied the temporal, spatial and processor locality of user and system references in parallel programs; and [Cher88] examined the behavior of parallel programs running on a VMP simulator, in particular the change in coherency overhead as block size was increased. The studies differ from this dissertation research in the type of experiment done and the workload used.

2.6. Critique of Previous Studies

Most previous analyses of coherency protocols have had weaknesses in methodology, both analytic technique and workload model, and choice of metric. Their methodology, be it case analysis, analytic modeling or parameterized simulation, restricted them to a static analysis of particular protocol functions or a simplified dynamic analysis. Case analysis is an important tool for specifying the functional completeness of a protocol, but it is a static modeling device. Analytic modeling and parameterized simulation are more sophisticated techniques. (For example, parameterized simulation characterizes dynamic protocol behavior.) However, they are only starting points and both have drawbacks.

Detailed analytic models with realistic parameter values are often computationally expensive to solve. They may be simplified in order to get a solution more quickly. The disadvantage of the simplified versions is that their aggregate behavior parameters and convenient (for solvability), but inaccurate, assumptions about the probability distributions do not model reality accurately. These simplifications have an important effect on the results of modeling coherency protocols.

In the absence of multiprocessor programs from which to determine workload parameters, all previous analytic studies (except [Owic89]) made certain assumptions about the type and frequency of sharing. They assumed that memory accesses to write-shared data are independent and uniformly distributed across processors. This was expressed in the models by a parameter for the proportion of accesses to write-shared data; its value was varied to emulate low or high levels of sharing.

The uniform access pattern models multiprocessor *contention* for write-shared data. Therefore the models' workload assumption guaranteed that those protocols that were designed to handle contention behaved well, relative to those that are more appropriate for a more sequential sharing behavior. It is well understood that uniprocessor programs exhibit temporal

and spatial locality in their memory reference behavior. (In fact, [Dubo82] and [Arch86] use the LRU stack model to generate references to private data and instructions.) There is no reason to assume a priori that the same locality principles will not apply to shared data.

Many previous studies have focused on total system performance, using processing power (i.e., multiprocessor utilization) as their metric. Processing power is a good summary metric, but it doesn't provide the detail needed either for a thorough analysis of the behavior of parallel programs or for multiprocessor cache and cache coherency protocol design. Different key aspects of system performance should be identified and analyzed separately. For example, measuring contention between the snoop and the CPU over use of the cache is important, because it may explain the cause of lower processor utilization and program speedup (as opposed to sequential portions of the code). Secondly, an understanding of cache miss ratio will aid in the design of cache organizations for multiprocessors; this is particularly critical if those designs must be based on different criteria than for uniprocessor systems. Finally, metrics other than processing power will provide a better understanding of the limits of bus-based systems. Bus utilization, and the effect of shared memory accesses on it, is important to measure, because the bus is the critical resource in single-bus multiprocessors.

In this dissertation I plan to correct the deficiencies of the previous work in several ways. First, a model of coherency overhead will be developed that incorporates a more accurate workload component, one in which the pattern of access to write-shared data is modeled in detail. Second, trace-driven simulation, using memory reference traces of parallel applications, will be used to obtain realistic parameter values for the model and to verify it. Third, the studies will analyze the causes of parallel program performance, focusing on the components of system throughput, rather than simply recording its level. These empirical studies analyze the cache and bus behavior of parallel programs and will also be done by trace-driven simulation. The combination of a more realistic workload and the detailed studies will provide a better analysis of the behavior of parallel programs, running on bus-based multiprocessors. Once that behavior

is determined, a more accurate measurement of coherency performance of particular protocols can be obtained. I treat the trace-driven methodology in detail in the next chapter; and the particular studies, their metrics and results, and comparisons to the other trace-driven work in Chapters 4 through 6.

2.7. References

[Agar88a] A. Agarwal and A. Gupta, "Memory-Reference Characteristics of Multiprocessor Applications under MACH", *Proceedings of the 1988 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 16, 1 (1988), 215-225.

[Agar88b] A. Agarwal, R. Simoni, M. Horowitz and J. Hennessy, "Scalable Directory Schemes for Cache Coherence", *Proceedings of the 15th Annual International Conference on Computer Architecture*, 16, 2 (May 1988), 280-289.

[Alle88] F. Allen, M. Burke, P. Charles, R. Cytron and J. Ferrante, "An Overview of the PTRAN Analysis System for Multiprocessing", *Journal of Parallel and Distributed Computing* (December 1988).

[Arch84] J. Archibald and J. Baer, "An Economical Solution to the Cache Coherency Problem", *Proceedings of the 11th Annual International Symposium on Computer Architecture*, 12, 3 (June 1984), 355-362.

[Arch86] J. Archibald and J. Baer, "An Evaluation of Cache Coherence Solutions in Shared-Bus Multiprocessors", *ACM Transactions on Computer Systems*, 4, 4 (November 1986), 273-298.

[Baro85] R. Baron, R. Rashid, E. Siegel, A. Tevanian and M. Young, "MACH-1: An Operating System Environment for Large-Scale Multiprocessor Applications", *IEEE Software* (July 1985).

[Bell85] C. G. Bell, "Multis: A New Class of Multiprocessor Computers", *Science*, 228 (April 1985), 462-467.

[Bita86] P. Bitar and A. M. Despain, "Multiprocessor Cache Synchronization: Issues, Innovations, Evolution", *Proceedings of the 13th Annual International Symposium on Computer Architecture*, 14, 2 (June 1986), 424-442 .

[Borr85] G. Borriello, S. Eggers, R. Katz, H. McKinley, C. Perkins, W. Scott, R. Sheldon, S. Whalen and D. Wood, "Design and Implementation of an Integrated Snooping Data Cache", Technical Report No. UCB/Computer Science Dpt. 85/199, University of California, Berkeley (January 1985).

[Bran85] W. C. Brantley, K. P. McAuliffe and J. Weiss, "RP3 Processor-Memory Element", *Proceedings of the 1985 International Conference on Parallel Processing* (1985), 782-789.

[Cens78] L. M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems", *IEEE Transactions on Computers*, C-27, 12 (December 1978), 1112-1118.

[Cheo88] J. Cheong and A. V. Veidenbaum, "A Cache Coherence Scheme With Fast Selective Invalidation", *Proceedings of the 15th Annual International Symposium on Computer Architecture*, 16, 2 (May 1988), 299-307.

[Cher86] D. R. Cheriton, G. A. Slavenburg and P. D. Boyle, "Software-Controlled Caches in the VMP Multiprocessor", *Proceedings of the 13th International Symposium on Computer Architecture*, 14, 2 (June 1986), 366-374.

[Cher88] D. F. Cheriton, A. Gupta, P. D. Boyle and H. A. Goosen, "The VMP Multiprocessor: Initial Experience, Refinements and Performance Evaluation", *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu, HA (May 1988), 410-421.

[Coff73] E. G. Coffman, Jr. and P. J. Denning, in *Operating Systems Theory*, Eaglewood Cliffs, New Jersey (1973), Prentice-Hall, Inc.

[Cytr88] R. Cytron, S. Karlovsky and K. P. McAuliffe, "Automatic Management of Programmable Caches", *Proceedings of the 1988 International Conference on Parallel Processing*, 2 (August 1988), 229-238.

[Dubo82] M. Dubois and F. A. Briggs, "Effects of Cache Coherency in Multiprocessors", *IEEE Transactions on Computers*, C-31, 11 (November 1982), 1083-1099.

[Edle85] J. Edler, A. Gottlieb, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, M. Snir, P. J. Teller and J. Wilson, "Issues Related to MIMD Shared-memory Computers: the NYU Ultracomputer Approach", *Proceedings of the 12th Annual International Symposium on Computer Architecture*, 13, 3 (June 1985), 126-135.

[Fran84a] S. J. Frank, "Synapse Tightly Coupled Multiprocessors", unpublished manuscript (1984).

[Fran84b] S. J. Frank, "Tightly Coupled Multiprocessor System Speeds Memory-access Times", *Electronics*, 57, 1 (January 12, 1984), 164-169.

[Full78] S. H. Fuller and S. P. Harbison, "The C.mmp Multiprocessor", Technical Report No. CMU/CS 78/146, Carnegie-Mellon University (October 1978).

[Good83] J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic", *Proceedings of the 10th Annual International Symposium on Computer Architecture*, 11, 3 (June 1983), 124-131.

[Gust82] R. N. Gustafson and F. J. Sparacio, "IBM 3081 Processor Unit: Design Considerations and Design Process", *IBM Journal of Research and Development*, 26, 1 (January 1982), 12-21.

[Hill86] M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. Ousterhout and D. A. Patterson, "SPUR: A VLSI Multiprocessor Workstation", *IEEE Computer*, 19, 11 (November 1986), 8-22.

[Karl86] A. R. Karlin, M. S. Manasse, L. Rudolph and D. D. Sleator, "Competitive Snoopy Caching", *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, Toronto, Canada (October 1986), 244-254.

[Karl88] A. R. Karlin, M. S. Manasse, L. Rudolph and D. D. Sleator, "Competitive Snoopy Caching", *Algorithmica*, 3 (1988), 79-119.

[Katz85] R. Katz, S. Eggers, D. Wood, C. L. Perkins and R. Sheldon, "Implementing a Cache Consistency Protocol", *Proceedings of the 12th Annual International Symposium on Computer Architecture*, 13, 3 (June 1985), 276-283.

[Lee87] R. L. Lee, P. Yew and D. J. Lawrie, "Multiprocessor Cache Design Considerations", *Proceedings of the 4th Annual International Symposium on Computer Architecture*, 15, 2 (June 1987), 253-262.

[Love88] R. Lovett and S. Thakkar, "The Symmetry Multiprocessor System", *Proceedings of the 1988 International Conference on Parallel Processing*, University Park PA (August 1988), 303-310.

[McCr84] E. McCreight, "The DRAGON Computer System: An Early Overview", *NATO Advanced Study Institute on Microarchitecture of VLSI Computers*, Urbino, Italy (July 1984).

[McGr86] S. McGrogan, R. Olson and N. Toda, "Parallelizing Large Existing Programs - Methodology and Experiences", *Proceedings of Spring COMPCON* (March 1986), 458-466.

[Neff86] L. Neff, "Clipper Microprocessor Architecture Overview", *Proceedings of Compcon '86*, San Francisco CA (March 1986), 191-195.

[Olso85] R. Olson, "Parallel Processing in a Message-Based Operating System", *IEEE Software* (July 1985), 39-49.

[Owic89] S. Owicki and A. Agarwal, "Evaluating the Performance of Software Cache Coherency", to appear in ASPLOS III. (April 1989).

[Papa85] M. S. Papamarcos and J. H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories", *Proceedings of the 11th Annual International Symposium on Computer Architecture*, 12, 3 (January 1985), 348-354.

[Pate82] J. H. Patel, "Analysis of Multiprocessors with Private Cache Memories", *IEEE Transactions on Computers*, C-31, 4 (April 1982), 296-304.

[Pfis85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture", *Proceedings of the 1985 International Conference on Parallel Processing* (1985), 764-771.

[Saty80] M. Satyanarayanan, "Commercial Multiprocessing Systems", *IEEE Computer*, 13, 5 (May 1980), 75-96.

[Sega84] Z. Segall and L. Rudolph, "Dynamic Decentralized Cache Schemes for an MIMD Parallel Processor", *Proceedings of the 11th International Symposium on Computer Architecture*, 12, 3 (June 1984), 340-347.

[Site88] R. L. Sites and A. Agarwal, "Multiprocessor Cache Analysis Using ATUM", *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu, HA (May 1988), 186-195 .

[Smit85] A. J. Smith, "CPU Cache Consistency with Software Support Using 'One Time Identifiers'", *Proceedings of the Pacific Computer Communications Symposium*, Seoul, Republic of Korea (October 1985), 142-150.

[Swea86] P. Sweazey and A. J. Smith, "A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE", *Proceedings of the 13th International Symposium on Computer Architecture*, 14, 2 (June 1986), 414-423.

[Tang76] C. K. Tang, "Cache System Design in the Tightly Coupled Multiprocessor System", *Proceedings of National Computer Conference* (1976), 749-753.

[Thac88] C. P. Thacker, L. C. Stewart and E. H. Satterthwaite, Jr., "Firefly: A Multiprocessor Workstation", *IEEE Transactions on Computers*, 37, 8 (August 1988), 909-920.

[Thak88] S. Thakkar, P. Gifford and G. Fielland, "The Balance Multiprocessor System", *IEEE Micro* (February 1988), 57-69.

[Veid86] A. V. Veidenbaum, "A Compiler-Assisted Cache Coherency Solution for Multiprocessors", *Proceedings of the 1986 International Conference on Parallel Processing* (August 1986), 1029-1036.

[Vern86] M. K. Vernon and M. A. Holliday, "Performance Analysis of Multiprocessor Cache Consistency Protocols Using Generalized Timed Petri Nets", *Proceedings of Performance '86 and ACM Sigmetrics 1986*, Raleigh NC (May 1986), 9-17 .

[Vern88] M. K. Vernon, E. D. Lazowska and J. Zahorjan, "An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-Consistency Protocols", *Proceedings of the 15th Annual International Symposium on Computer Architecture*, 16, 2 (May 1988), 308-315.

[Widd80] L. C. Widdoes, Jr., "The S-1 Project: Developing High-Performance Digital Computers", *Proceedings of Compcon 80*, San Francisco CA (February 1980), 282-291.

[Yen82] W. C. Yen and K. S. Fu, "Analysis of Multiprocessor Cache Organizations with Alternative Main Memory Update Policies", *Proceedings of the 8th Annual International Symposium on Computer Architecture*, 9, 3 (May 1982), 89-100.

[Yen85] W. C. Yen, D. W. L. Yen and D. S. Fu, "Data Coherence Problem in a Multicache System", *IEEE Transactions on Computers*, C-34, 1 (January 1985), 56-65.

3 Methodology

3.1. Introduction

The goal of this research is to understand the sharing behavior of parallel programs and to analyze its impact on multiprocessor cache and bus performance, given particular snooping coherency protocols. One of the unique and important aspects of the work is that it was intended that the studies be driven by a *real* workload. Therefore appropriate parallel programs had to be found, and memory reference traces had to be collected from them in order to carry out the experiments. Several problems were encountered in doing this, all relating to the parallel nature of the workload.

First, the programs themselves were difficult to locate. Not many substantial parallel programs had been written, and most of those in existence were proprietary to particular multiprocessor companies or customer-owned, and therefore unavailable to the research community. In addition, in order to avoid justifiable criticisms of an inappropriate workload, I considered it

crucial to use programs that had been written for a multiprocessor architecture similar to the one being studied. Therefore I limited the search to those running on bus-based, shared memory machines. The programming paradigm of these programs was a model in which the granularity of parallelism was a process. This eliminated the relatively larger body of numerical programs that had been written for other multiprocessor architectures, and whose granularity of parallelism was much finer, on the level of do loop iterations, for example.

Even when using programs written for bus-based, shared memory multiprocessors, there were still questions about the applicability of memory reference traces generated on one machine being used for simulations of another. Issues concerning inter-processor synchronization and varying instruction execution times had to be resolved. The resolution necessitated extensive postprocessing of the traces, to identify shared accesses and synchronization points. Postprocessing expanded the traces to sizes that were prohibitive for disk residence. The storage problems were aggravated by the multiprocessor nature of the experiments, i.e., that separate traces were needed for each processor. Therefore trace compaction techniques, specially designed for handling parallel trace content, had to be developed.

Special problems were encountered in debugging the multiprocessor simulator, because of the asynchronous activity of the processors. The list of potential processor interactions is far too numerous to allow individual testing. Therefore techniques were developed to trap system-wide errors as they occurred in the actual simulations, to prevent the incorrect actions of one processor from perturbing the behavior of others.

Each phase of the methodology: trace generation, trace postprocessing and compression, and the simulation itself, will be treated in detail in the subsequent chapter sections. Each section will address the problems encountered and discuss the solutions.

3.2. Trace-driven Simulation

All studies in this dissertation are performed by trace-driven simulation. Trace-driven simulation has the advantage over other forms of modeling in that its input, an address trace, exactly characterizes the behavior of the program (or a portion of the program) from which it was generated. Therefore the order and frequency of events, in this case the coherency-related operations, can be accurately measured and analyzed. To the extent that the program is representative of a "typical" workload and the details of the system are sufficiently simulated, the simulation results are an accurate portrayal of system behavior. Its advantage over measuring activity via a hardware monitor is that one can change configuration parameters to do comparative studies.

The drawbacks of trace-driven simulation stem primarily from the traces being a worm's eye view of an actual workload. First, because of storage and simulation time constraints, a relatively small amount of activity can be simulated. Second, traces that include operating systems activity are difficult to obtain. A common technique for generating traces is through software that behaves like a symbolic debugger, breakpointing at key locations and dumping trace information (see section 3.3). It is difficult to use this type of trace generator in conjunction with operating systems code, because of kernel protection and the inability to recompile the operating system. The lack of operating systems references skews the results by eliminating the effect of context switching and the (presumably) lower locality of reference. These perturbations are important per se, and also because in many systems the operating system dominates the workload. The only other set of multiprocessor traces overcomes this disadvantage, through the use of a microcode generation technique in which samples of memory reference activity are gathered across several contexts [Site88].

An additional reason for doing trace-driven simulation of parallel programs is that it will yield parameter values that can be used in analytic models of multiprocessor activity. Research

in parallel architectures is still in such a stage of infancy that we do not yet have good intuition for parameter values of sharing behavior. (A more complete critique of trace-driven simulation appears in [Clar83, Smit85].)

3.3. The Traces

Trace-driven simulation has traditionally been applied to uniprocessor studies. Recent advances in parallel computing have provided an opportunity to do simulations of multiprocessors. In particular, the emergence of commercial multiprocessors and the development of parallel algorithms to run on them have made traces of parallel programs available. For the studies in this dissertation, traces were generated from four parallel programs. The programs are all CAD tools that were developed for single-bus, shared memory multiprocessors (see Table 3-1). The choice of application area was deliberate, so that the workload being analyzed was appropriate for small-scale machines. (See section 3.5.1 for a discussion of this architecture.) One program is production quality (SPICE); the others are research prototypes. Two of the programs (CELL and TOPOPT) are based on simulated annealing algorithms. CELL [Caso86]

Parallel Applications			
Trace Name	Architecture, Operating System	Program Description	Number of Processors
CELL	Sequent Balance, Unix	simulated annealing algorithm for cell placement	12
TOPOPT	Sequent Balance, Unix	simulated annealing algorithm for topological optimization	11
VERIFY	Sequent Balance, Unix	logic verification	12
SPICE	ELXSI 6400, Embos	direct method circuit simulator	5

Table 3-1: Traces Used in the Simulations

The traces used in the sharing simulations were gathered from parallel programs that were written for shared memory multiprocessors. The programs are all "real", being either production quality (SPICE) or research applications.

uses a modified simulated annealing algorithm for IC design cell placement that attempts to minimize total area and wire length. The algorithm allows cells to overlap in the early stages and finally removes all overlaps by the time it completes. The cell descriptions reside in shared memory, and all cell moves (for example, placement within a processor's chip area or exchanging cells with another processor) are generated and accepted independently. For the trace, the program placed twenty-three cells. Program speedup is 6 on an 8 processor Sequent Balance 8000, with comparable results to uniprocessor implementations.

TOPOPT [Deva87] does topological compaction of MOS circuits. The circuit is represented in symbolic form (as a Weinberger Array); the algorithm minimizes the layout by repeatedly folding the rows of the array. Representations for the array, gates and signals reside in shared memory. Dynamic windowing results in array sharing (windows into the array change processors over time), and dynamic partitioning shares gates and signals through inter-window exchanges. The input was a technology-independent multi-level logic circuit. The program achieves a speedup of 6 on an 8 processor Sequent Balance 8000, while generating solutions similar in quality to uniprocessor results.

VERIFY [Ma87] is a combinational logic verification program, which compares two different circuit implementations to determine whether they are functionally (Boolean) equivalent. The algorithm uses a two-phase technique: the enumeration phase lists all inputs that will produce outputs of either zero or one for the first circuit, using a PODEM-based [1] enumeration algorithm; the second phase simulates the inputs of the first circuit on the second and compares the two sets of outputs. The trace snapshot is taken from the enumeration phase, and the major shared structure is the graphical representation of the first circuit. The input for the trace was a combinational benchmark circuit that is used for evaluating different test generation algorithms [Brgl85]. The program achieves a speedup of 7.8 on an 8 processor Sequent Balance 8000, 10

[1] "path-oriented decision making" (depth-first search of graphs representing the circuits).

to 11 on a 12 processor machine.

The final program, SPICE [McGr86], is a circuit simulator; it is a parallel version of the original direct method approach. The algorithm solves a set of nonlinear ordinary differential equations, which are a description of the circuit's devices. The ODE's are integrated to yield a set of nonlinear algebraic equations. These equations are then solved iteratively using the Newton-Raphson technique, which first linearizes the equations and then solves the resulting set of sparse linear equations using LU-decomposition. When executing on an ELXSI 6400, this program achieves speedup that is almost linear with the number of processors, across a wide variety of inputs. For this trace the program's input was a chain of 64 inverters.

All applications use a coarse-grain parallel programming paradigm for carrying out the parallel activities (see Figure 3-1). The granularity of parallelism is a process, in this case one for each processor in the generation machine. The model of execution is single-program-multiple-data, with each child process independently executing identical code on a different portion of shared data. The shared data are divided into units that are placed on a logical queue in shared memory. Each process takes a unit of work from the queue, computes on it, writes results, and then returns the unit of work to the end of the queue. When the programs first begin execution, there is unusual contention for the locks protecting the queue of work, since all child processes try to take a unit of work simultaneously. However, only one process will obtain access to the queue at a time. Assuming that each process does a comparable amount of processing, they will thereafter access the queue in the same order and spaced in time by the computation interval. This self-scheduling is disrupted by synchronization barriers,[2] which are used to separate phases in the computation. The disruption causes more busywaiting and therefore an increase in references to shared addresses. All four programs followed this basic procedure,

[2] Synchronization barriers are synchronization points in the application that cause all parallel processes that have reached them to wait until the other processes have arrived. Then all processes proceed with the execution simultaneously.

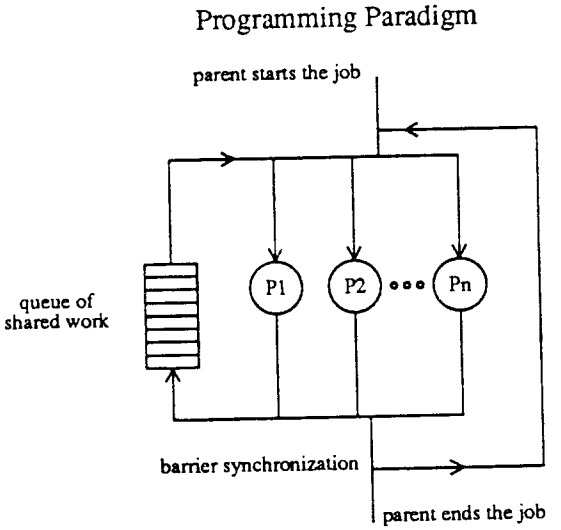

Figure 3-1: Flow Chart of the Programming Paradigm of the Parallel Traces

This simplified representation illustrates the programming paradigm of the parallel programs under study. The parent process starts and ends the program, and forks child processes that do the parallel portion of the computation. Each child process executes the same code. At certain points in the parallel computation, the children resynchronize, and then repeat the computation. Within each iteration, the children process different portions of the work queue, which resides in shared memory. For example, in a parallelized circuit simulator, the circuit would be divided into groups of devices (nodes). In each iteration, each child would process a particular node. Data sharing occurs because the inputs and outputs of the nodes interconnect, and a node may be processed by different child processes in different iterations.

with two exceptions: TOPOPT does not use locks to protect the write-shared data, and there are no synchronization barriers in VERIFY.

The scope of the traces is limited to memory references of the applications, and the operating system runtime routines used to set up shared memory and support locking. Because of the well known difficulty in tracing operating systems code (see section 3.2), the path of the applications through the rest of the operating system is not captured in the traces. In addition, each of the parallel processes was run on a single processor without process migration. Despite these omissions, the simulations should produce reliable results. The only trace-driven study that included operating systems activity found that sharing in the operating system added little

to the user figures [Agar88]. For example, the proportion of write-shared references to total references and shared data references to total data references remained roughly constant when operating systems references were included. Process migration exacerbates bus bandwidth demands by causing additional misses when faulting in the process on a new processor and by creating coherency bus traffic for private data or cache flushes to prevent it. [Agar88] found evidence that process migration decreased the temporal locality of shared references, i.e., increased contention for shared data, and introduced sharing for private data. Since I was interested in measuring the amount of bus traffic to shared data only, unperturbed by the effects of process migration, I chose not to emulate process migration in the simulations.

Both the Sequent and ELXSI traces were generated using a software trace generator.[3] Both generators function like symbolic debuggers, using trace-trap facilities to halt at each instruction and dump trace information, both for instructions and their operands. They also included the ability to start and stop tracing and to determine the address range of the code and data sections, including the subrange for the shared variables. (The latter was needed for trace verification and the identification of shared data.) The ELXSI generator was itself a parallel program. Each child process executed on a different processor, tracing the process of the parallel application that had been scheduled on its processor. The traces were generated on a per processor basis, with the trace records of each processor outputted to a separate output device. The Sequent tracer was much more primitive. It executed instructions from each processor on a round robin basis; memory references from all processors were sent to a single output file and had to be separated during the postprocessing phase.

[3] The Sequent tracer that was used was adapted from a version written at Sequent; the ELXSI tracer was written at ELXSI by John Sanguinetti.

3.4. Trace Postprocessing

Accurate trace-driven simulation of parallel programs requires changes to the traditional trace-driven methodology. Both the traces and the simulator are affected. The traces must be postprocessed to detect shared data and synchronization points, and the sheer volume of data must be handled through special compaction techniques. The simulator must be able to detect program development errors in the coherency protocols that are caused by the asynchronous interaction of the multiprocessor components. These errors occur both within a single processor node (between the snoop and the portion of the cache controller that acts on behalf of the CPU) and across processors. (The changes to trace handling will be discussed in the following two subsections; enhancements to the simulator appear in section 3.5.4.)

3.4.1. Detecting and Processing Sharing in the Parallel Traces

In both the Sequent Balance 8000 and the ELXSI 6400 all child processes that execute in parallel have their own virtual address space. If a multiprocessor simulator that implemented the distributed, hardware protocols maintained coherency merely on the basis of a cache tag comparison, coherency operations would be generated for private data, as well as shared. Therefore shared references must be explicitly identified during the trace postprocessing phase that precedes simulation. The identification ensures an accurate coherency enforcement, and has the beneficial side effect that shared references can be detected for separate analysis. For this purpose, shared accesses were further classified as locks or the applications shared data that was protected, and cacheable or noncacheable. The identification was achieved through symbol tables, load (memory) maps and interactive (generation) runtime identification. In SPICE all shared data was grouped into separate Fortran COMMON blocks that were declared to reside in shared memory via system calls; the symbol table identified the starting address and length of these blocks. Load maps contain addresses of procedures and data of the applications and the library routines that support shared memory and lock manipulation. They were used to separate

references to the stack, instruction space, and certain areas within the heap that are known system shared areas, as well as to locate entry points to specific routines that handled, for example, locking. Other addresses had to be identified during runtime. Examples are data that was allocated dynamically into shared memory (locks and semaphores in SPICE and all shared data in the Sequent-generated programs) and key code sequences that were embedded within a subroutine, rather than being a discrete function (such as sequences for locking, unlocking, barrier synchronization and the start and end of main algorithm iterations[4]). For the latter both the starting address of the code sequence and the instructions contained within it were needed. Once addresses of shared data and synchronization code were known, the traces were postprocessed to detect and flag all occurrences. This made detection by the simulator trace-independent.

The onset of lock and unlock sequences in the traces was flagged, so that serial access to shared areas could be enforced and busywaiting could be implemented (and measured) in the multiprocessor simulations. In addition, a common lock/unlock sequence was embedded in the simulator. It was used in the simulations in lieu of the locking algorithms implemented on the ELXSI and Sequent machines, so that the sharing statistics were not perturbed by the differences between the algorithms. Therefore all locking sequences in the traces were marked, through the dynamic probing described above, to prevent their being processed by the multiprocessor simulator.

Synchronization among the processes is dependent both on the sharing exhibited by the algorithms of the program, and on particular architectural features being simulated, such as the size of the caches and the cache hit and miss times. Because of the latter factor, the synchronization mechanisms (locking, unlocking and synchronization barriers) are implemented as part of the simulator. The order in which processes obtain locks and reach barriers, and the frequency and length of busywaiting for each type of synchronization, is therefore determined by

[4] See a description of the programming paradigm in section 3.3.

the dynamic behavior of the simulated processors. The exact order in which processors obtained locks and reached barriers on the generation machine is not used, and, consequently, busywait sequences for both were stripped from the traces. Busywaiting for locks and barriers is simply repetitions of code sequences for obtaining the lock and checking the barrier flag. Both could be detected through interactive runtime identification.

In the ELXSI-generated program, SPICE, coherency was maintained via software methods. Since I intended to study distributed, hardware coherency techniques, all memory references reflecting the software implementation, such as cache flushing instructions, were also eliminated from the traces. These were identified during postprocessing by pattern matching on specific opcode values that had been detected at runtime.

In addition to the postprocessing needed specifically for sharing, routine consistency checks were done on both addresses and instructions. The checks insured that (1) all addresses were valid for their type (instruction or data), (2) data addresses were in the proper stack, heap or constant address ranges, as appropriate; (3) instructions contained the proper number of operands for their addressing modes; and (4) opcodes were legal for the type of instruction. After postprocessing, the traces contained the address of the memory reference, the type of reference (instruction, load operand, store operand), its shared code (for data: private data, a lock or applications shared data, cacheable or noncacheable; for instructions: the beginning and end of a lock or unlock routine, the beginning of a busywaiting sequence, a barrier, or a coherency-related instruction which should be ignored by the simulator), whether the reference was from the user program or the runtime library, the number of execution cycles on the generation machine and the opcode, both only for instructions.

Finally, both the Sequent and the ELXSI have variable length instructions and data. The number of bytes actually transferred during the memory accesses was also determined during postprocessing, based on the particular opcode and data type. Therefore postprocessed traces

contained additional memory references for all accesses that were larger than four bytes. This provided correct processing when varying the cache block size in the simulations and made the traces independent of the particular block and word size of the generation machine.[5]

3.4.2. Trace Compaction Using a Cache Filter for Parallel Programs

An additional problem of trace-driven simulation is the large quantity of disk storage required for the traces. This amount can be excessive even in uniprocessor systems, because a fairly large snapshot (in numbers of memory references) is needed to obtain statistically significant results. As cache sizes increase, this number mushrooms. The problem is exacerbated in multiprocessor simulations, because the size of traces for parallel programs is directly proportional to the number of processors being simulated. For example, the 6 million reference (per processor) traces used in this dissertation, after postprocessing, comprised approximately 1.8 to 2 gigabytes for each 11 or 12 processor Sequent trace and .9 gigabytes for the 5 processor SPICE. Practically speaking, the traces must be compacted to be usable. Traditionally, encoding schemes, such as Ziv-Lempel compression [Ziv78], have been used. Memory reference traces that are used for cache studies can be further reduced by special techniques, such as cache filtering.

Uniprocessor cache filters reduce the size of memory reference traces by removing all cache hits from the trace. The filter is a cache simulator, whose input is the original trace, and which outputs trace records of cache misses and summary information of the missing cache hits. In other words, only those references that cause bus operations are explicitly recorded in the filtered trace. There are two restrictions on the configurations of caches that are analyzed with the filtered traces. Both restrictions guarantee that caches of varying sizes see a correct hit/miss usage of references. First, the analyzed caches must contain no fewer sets than the cache simulated in the filter; second, they must use the same block size as the filtered cache.

[5] The word size on the ELXSI 6400 is 64 bytes.

The stack deletion filtering technique, described in [Smit79], was developed to analyze program paging behavior, but can easily be adapted for cache studies. In the most general version of the scheme references to the first D-1 positions (D being the deletion parameter) in an LRU stack of memory references are removed from the trace, and a counter, reflecting the amount of processing time required for the number of references that were eliminated, is outputted with the next recorded reference. For one million reference traces, the technique achieves a reduction in trace length (where length is defined in numbers of entries) of a factor of 14 to 36, when D is, for example, 6. However, since the algorithm is not directly tied to a cache simulation, some error in the hit/miss classification is introduced when using the compressed data, when D is greater than 2. The error should decline as cache size increases.

[Puza85] represents the summary cache hit information with runlengths of consecutive hits. For caches ranging from 4K to 16K bytes, with 64 and 128 byte blocks, his method produces a filtered trace approximately one tenth the size of the original. My technique for parallel traces was adopted from this approach.

The compression technique in [Samp89] records the difference between the address of the reference that hits in the cache and the one that most recently missed in the same block. Because it exposes the patterns of locality in memory references, it produces traces that are good candidates for further compression by the schemes that rely on pattern matching techniques. When used as a preprocessor to Ziv-Lempel compression, the technique produces compressed files at least as small as [Puza85]. Although it does not compact as well as other schemes, it retains all information from the original trace. Therefore the original trace can be reconstructed, for example, to regenerate traces for a different block size or smaller cache size by one of the more tightly compacting methods.

Because of the additional bus operations caused by sharing, I broadened the criteria for reference elimination by cache filters for traces of parallel programs from Puzak's simple

hit/miss model to a filtering technique that is based on any change of state. In this more general scheme, state is defined as the superset of cache state, dirty/clean state, coherency state and synchronization state.[6] Cache state is the criteria used in [Puza85]; it differentiates between valid and invalid blocks, and is required to support the hit/miss criteria. Dirty/clean state distinguishes between the first write to a block and all others. It is needed for two reasons. First, only the first write in a sequence of per processor writes generates a bus operation (the invalidation signal) in the write-invalidate protocols; and, second, dirty, private data must be copied to memory on block replacement.

Coherency state includes the five MOESI values (invalid, private clean, shared clean, private dirty, shared dirty). Generating a memory reference for any potential change of coherency state essentially means that all shared operands are outputted. All write-shared references must be recorded, because it cannot be determined a priori, which will result in a bus broadcast in the write-broadcast protocols. Read-shared operands must also be included, because in the write-invalidate protocols, write hits produce a different bus operation (an invalidation) than write misses (a full data transfer). For the hit to be detected, the block must already reside in the cache when the write occurs.

Synchronization state comprises a processor's first attempt to obtain a lock, its acquiring the lock, its unlocking it, its reaching a barrier or flushing the cache. Memory references that correspond to these coherency-related instructions (i.e., instructions that implement locking and unlocking, reaching a barrier, and executing software coherency mechanisms) must also be outputted. If not explicitly recorded in the filtered trace, the multiprocessor simulator would be unable to ignore and/or replace them with other sequences of code (see section 3.4.1).

---

[6] [Thom88] uses a similar state definition as the basis for a technique for simulating multiple sized caches in a multiprocessor and shows that certain aspects of the definition (dirty/clean state and coherency state) obey the cache inclusion property. ([Matt70] showed that cache state, i.e., validity, obeyed inclusion in their development of the stack analysis technique for analyzing cache behavior.) It is this inclusion property that allows multiple sized caches to be simultaneously simulated or filtered.

Explicitly specifying these output records leaves runlengths only to hits of instructions and private data (excepting the first write to the blocks).[7] The amount of trace reduction achieved by substituting runlengths for trace records is called the compression ratio. The compression ratio is defined as the number of items in all runlengths (number of references eliminated), divided by the total number of memory references. The compression ratio for 6 million reference versions of the traces in this dissertation, using a 16K byte cache filter with 4, 8, 16 or 32 byte blocks, ranged from .82 to .85 for CELL, .72 to .86 for SPICE, .85 to .89 for TOPOPT and was .86 for VERIFY. Expressed in the inverse terminology, the filtered traces were, on the average, 15 percent of the original, unfiltered traces. When further compressed with Ziv-Lempel encoding, the final traces were approximately 4.5 percent of the originals.

3.5. The Multiprocessor Simulator

3.5.1. Its Underlying Architecture

The parallel simulator (named "charlie", after Snoopy's well known master) emulates a simple, shared memory architecture, in which a modest number of processors (five to twelve) are connected on a single bus. The CPU design is RISC-like [Patt85], assuming one cycle per instruction execution. Not all instructions follow this model, e.g., multiply and divide; therefore the bus utilization results (Chapters 5 and 6) will be slightly overestimated and throughput underestimated (Chapter 6), because the simulation processors return to use the bus more quickly than in a real machine. All other metrics used in the studies, for example, cache miss rates and numbers of bus operations, should be unaffected. Since each processor executes

[7] The runlengths were further subdivided into separate runs of contiguous reads and writes. The subdivision was required to ensure correct simulation, because the simulator's cache controller design implemented one-cycle cache reads and two-cycles cache writes.

almost identical code[8], assuming faster times for a small subset of the instructions should slow down all processors uniformly. Therefore the order of system-wide shared references should remain approximately the same as with simulations that follow the cycle times of the generation machine.[9]

With the exception of those cache parameters that are varied in the studies, the memory system architecture of the simulator is roughly that of the SPUR multiprocessor [Hill86]. The simulator has a one-level cache, on the board; it is direct mapped, with one-cycle reads and two-cycle writes. There are two copies of the tag and state, one for the CPU, the other for the snoop. Its cache controller implements segment-based addressing, no fetch-bypass on reads, a test-and-test-and-set sequence for securing locks [Wood87], and many of the timing constraints of the actual SPUR implementation. Bus arbitration is implemented using a modified NuBus protocol [Gibs88], and bus contention is accurately modeled. Several of the architectural specifications are stipulated at runtime to allow flexibility in changing the studies; examples are the choice of coherency protocol, the cache configuration (cache size, block size and associativity), the number of processors, and timing specifications for bus operations, bus arbitration, and cache controller and snoop functions. The activities of the cache controller and bus are implemented in fine detail, with separate timing variables for most suboperations; the CPU is a black box; and, since no program results are kept, main memory and the data portion of the cache are nonexistent.

3.5.2. Implementation of the Simulator

The simulator is constructed as a group of lightweight tasks, executing within a single process. Each task has its own stack, which is copied into the stack space of the process, when

---

[8] Except for those processing either the first or last iteration of a loop.

[9] Simulations incorporating the cycle times of the ELXSI 6400 and Sequent (National Semiconductor 32000) processors were run for comparison to the results in Chapter 5. Except for bus utilization, which was lower, all results were consistent with the ones reported in this dissertation.

it is its turn to be executed. The tasks are made to look as though they are running in parallel by manipulating both systemwide and task-specific clocks (described below). Each task simulates a component of a multiprocessor, (e.g., a CPU, the processor's cache controller (PCC), the snooping portion of the cache controller or the bus). A deterministic event-driven simulator base, Simon [Fuji83, Hell84], handles all task scheduling, and synchronization and message passing among the tasks. For example, a message may be a cache controller request to the bus to read a block of data. The bus will not respond to the message until it has finished the current bus transaction, and the cache controller is next in line according to the NuBus protocol.

There are two sets of clocks in the simulator. The global clock indicates the current time in the multiprocessor system as a whole. In addition, each task has its own clock that is incremented to reflect the amount of time taken by a particular function, such as a cache lookup or bus arbitration. All tasks are scheduled by comparing the tasks' private clocks to the global clock, and then scheduling the task with the minimum private clock value. In the architecture and coherency protocol-independent simulations used in the sharing model (described in detail in Chapter 4), the clocks were incremented by a constant value for each memory reference (imitating round robin scheduling of instructions). In the architecturally detailed and protocol-specific simulations for the studies in Chapters 5 and 6, the increments accurately mimicked the asynchronous behavior of a multiprocessor system.[10]

3.5.3. Using the Traces

The traces were generated on a per processor basis. The number of processors in the simulations is identical to the number of processors used in trace generation. For SPICE this number is 5, and for the Sequent traces either 11 or 12. Each processor trace is a separate input stream to the simulator. As described in section 3.4.1, synchronization among the separate

[10] Other multiprocessor simulators use round robin scheduling even for realistic simulations, e.g., WASHCLOTH [Gott80].

input streams depends on the use of locks and barriers in the the programs and is handled directly by the simulator.

To avoid cold start effects in the simulations, all caches obtain steady state before statistics are gathered. Steady state was determined for each trace separately;[11] simulation statistics were then gathered on the next 300,000 references per processor (approximately). Several longer trace snapshots (one and two million references) for SPICE were analyzed for comparison to the 300,000 reference data. Most relevant metrics (for example, cache miss ratio and the number of bus operations for shared data) were stable across the different sample sizes; those that changed accounted for such a small percentage of total performance that their increased value was still of no consequence. For example, the percentage of per-processor busywaiting cycles within total cycles increased by a third. However, they originally accounted for only .2 percent of total cycles; therefore the increase was a .06 percent gain in total cycles.

The simulator keeps per processor and system-wide statistics on memory references, misses, bus operations, bus arbitration delay, coherency-related CPU delay, snoop operations, consecutive writes to shared addresses, interprocessor contention for shared addresses and busywaiting for locks and barriers, both in numbers and cycles consumed. Each category is broken down into appropriate subcategories (for example, type of memory reference or type of snoop operation); each subcategory is further subdivided into separate statistics for locks and applications shared data.

[11] The technique for determining the onset of steady state is similar to that used in [Site88]. I first constructed a plot of cumulative first reference misses, (i.e., those misses that occur for empty cache locations), versus time, where time was measured in numbers of memory references. The plots were taken over a six million reference trace for one processor of each program. Steady state occurs when the rate of first reference misses drops sharply. For the studies in this dissertation that point was defined to be when the first reference miss rate over the next 300,000 references (the trace snapshot for all studies) became .002 or less.

3.5.4. Multiprocessor Debugging Techniques

The asynchronous nature of generating and satisfying memory requests in a multiprocessor make debugging more difficult than for uniprocessor simulators. For example, errors introduced to the state of a cache block by one processor may not be detected until that address's use by another processor. The amount of time between these events can be considerable (typically tens of thousands of cycles in these simulations). Two techniques were developed to streamline the asynchronous activities and catch inter-processor errors.

First, protocol-specific cache controller and snoop operations in the multiprocessor simulator were table-driven. The tables provided a mapping of legal inputs (the type of memory request, the current state of the block, etc) to correct outputs for carrying out all aspects of satisfying memory references (such as the bus operations, state changes and other actions of the snoops and PCCs). They also guaranteed that illegal combinations of inputs were detected. Code coverage techniques were applied to the tables to insure that each operation, i.e., that each cache state transition, executed correctly. The use of the tables also simplified the code for executing shared memory references, both for the operations themselves, such as a bus transaction, and for the side effects, such as snoop interference with a processor request.

Second, system-wide assertions of correctness were provided at certain points in the simulation. Embedded into the simulator was a table of acceptable global cache state configurations, given the bus operation that had just taken place on the backplane. For example, after a data read that invalidates other caches under the Berkeley Ownership protocol, only one cache should hold the block private and dirty; in all other caches, it should be invalid. After each bus operation, the current state of all caches for the address of the operation was checked for a match with one of the assertions. Any coherency protocol violations detected by the assertions halted simulation.

3.5.5. Summary

All studies in this dissertation were performed by trace-driven simulation. The methodological cycle from trace generation to data analysis was as follows. First, memory reference traces were generated from four parallel programs. Second, the raw trace output was postprocessed to identify shared variables and synchronization points and to eliminate those references that were specific to the generation machine. Third, the postprocessed traces were compressed, using a cache filter specially designed for traces of parallel programs. The final step was the simulation itself, which included statistics gathering and analysis.

3.6. References

[Agar88] A. Agarwal and A. Gupta, "Memory-Reference Characteristics of Multiprocessor Applications under MACH", *Proceedings of the 1988 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 16, 1 (1988), 215-225.

[Brgl85] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran", *Proceedings of the 1985 IEEE International Symposium on Circuits Systems*, Kyoto, Japan (June 1985).

[Caso86] A. Casotto, F. Romeo and A. Sangiovanni-Vincentelli, "A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells", *Proceedings of the IEEE International Conference on Computer-Aided Design*, Santa Clara, CA (November 1986), 30-33.

[Clar83] D. W. Clark, "Cache Performance in the VAX-11/780", *ACM Transactions on Computer Systems*, 1, 1 (February 1983), 24-37.

[Deva87] S. Devadas and A. R. Newton, "Topological Optimization of Multiple Level Array Logic", *IEEE Transactions on Computer-Aided Design* (November 1987).

[Fuji83] R. M. Fujimoto, "SIMON: a Simulator of Multicomputer Networks", Technical Report No. UCB/Computer Science Dpt. 83/140, University of California, Berkeley (September 1983).

[Gibs88] G. A. Gibson, "SpurBus Specification", to appear as Computer Science Division Technical Report, University of California, Berkeley (December 1988).

[Gott80] A. Gottlieb, "WASHCLOTH - The Logical Successor to SOAPSUDS", Ultracomputer Note #12, Courant Institute, NYU (December 1980).

[Hell84] D. E. Heller, "Multiprocessor Simulation Program SIMON", Shell Development Company (November 1984).

[Hill86] M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. Ousterhout and D. A. Patterson, "SPUR: A VLSI Multiprocessor Workstation", *IEEE Computer*, 19, 11 (November 1986), 8-22.

[Ma87] H. T. Ma, S. Devadas, R. Wei and A. Sangiovanni-Vincentelli, "Logic Verification Algorithms and their Parallel Implementation", *Proceedings of the 24th Design Automation Conference* (July 1987), 283-290.

[Matt70] R. L. Mattson, J. Gecsei, D. R. Slutz and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies", *IBM Systems Journal*, 9, 2 (1970), 78 - 117.

[McGr86] S. McGrogan, R. Olson and N. Toda, "Parallelizing Large Existing Programs - Methodology and Experiences", *Proceedings of Spring COMPCON* (March 1986), 458-466.

[Patt85] D. A. Patterson, "Reduced Instruction Computers", *Communications of the ACM*, 28, 1 (January 1985), 8-21.

[Puza85] T. R. Puzak, *Analysis of Cache Replacement Algorithms*, PhD Thesis, University of Massachusetts, (February 1985).

[Samp89] A. D. Samples, "Mache: No-Loss Trace Compaction", *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* (1989).

[Site88] R. L. Sites and A. Agarwal, "Multiprocessor Cache Analysis Using ATUM", *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu, HA (May 1988), 186-195 .

[Smit79] A. J. Smith, "Two Methods for the Efficient Analysis of Memory Address Trace Data", *IEEE Transactions on Software Engineering*, SE-3, 1 (January 1979), 94-101.

[Smit85] A. J. Smith, "Cache Evaluation and the Impact of Workload Choice", *Proceedings of the 12th Annual International Symposium on Computer Architecture*, 13, 3 (June 1985), 64-73.

[Thom88] J. F. Thompson, "Efficient Analysis of Caching Systems", Computer Science Division Technical Report No. UCB/374, University of California, Berkeley (Spring, 1988).

[Wood86] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, J. Pendleton, S. A. Ritchie, G. S. Taylor, R. H. Katz and D. A. Patterson, "An In-Cache Address Translation Mechanism", *Proceedings of the 13th Annual International Symposium on Computer Architecture*, Tokyo, Japan (June 1986), 358-365.

[Wood87] D. A. Wood, S. J. Eggers and G. A. Gibson, "SPUR Memory System Architecture", Technical Report No. UCB/Computer Science Dpt./87/394, University of California, Berkeley (December 1987).

[Ziv78] J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-rate Coding", *IEEE Transactions on Information Theory*, 24, 5 (September 1978), 530-536.

4 The Write Run Model

4.1. Introduction

In this chapter I analyze the sharing behavior of the four parallel programs. I was interested in determining both (1) the *amount* of write sharing in the applications, and (2) the *pattern* of multiprocessor accesses to the write-shared data, i.e., whether there was inter-processor contention for the shared data *(fine-grain sharing)* or whether each processor accessed it multiple times before another processor intervened *(sequential sharing.)* Both the quantity and pattern of sharing are important factors in relative coherency protocol performance. The emphasis here is on the write sharing that is inherent in the application programs themselves, rather than that caused by the underlying memory system architecture, or the cache coherency protocol. To this end, the sharing study was conducted as independent of the architecture, implementation, and coherency protocol as possible. Its most important generality was basing the analysis on the one-word unit of access of the CPU, rather than a specific cache block size.

The primary reason for examining sharing behavior is to evaluate the performance of cache coherency protocols. I have chosen to compare examples of write-invalidate and write-broadcast protocols, both of which are subsets of the MOESI classification. The specific types of protocol were selected, because they are polar opposites in terms of their approach to maintaining coherency on bus-based multiprocessors, and they have been widely published and implemented (see Chapter 2).

To complete the protocol evaluation I first develop a simple (and architecture-independent) model of write sharing, whose parameter values are derived from the sharing analysis. The model is based on the inter-processor sharing activity, and reflects the costs of write sharing under the two protocols. I then compare the model's approximations of protocol performance to the results of realistic multiprocessor simulations, in which the cache architecture (particularly block size) and coherency protocol specifications of the simulator are very detailed. Results indicate that the model is a good predictor of protocol performance for the write-broadcast protocols. This is primarily because the one-word *coherency block*[1] of these protocols matches the unit of analysis in the sharing study. However, for write-invalidate the model is not accurate, because its coherency block is sized to the larger cache block. Write-invalidate's performance is quite sensitive to the shared data reference pattern within the block. When the pattern is one of sequential sharing, with good per-processor locality of reference, coherency overhead is much lower than for patterns that exhibit fine-grain sharing. By limiting the analysis to the one-word access of the CPU, the architecture-independent model does not capture a processor's spatial locality of reference and therefore mispredicts write-invalidate's coherency overhead. Incorporating the size of the coherency block into the model produces more accurate results.

---

[1] The coherency block is that portion of the cache block that is effected by a coherency operation. For the write-invalidate protocols it is the entire cache line; for write-broadcast, one word.

I shall begin the chapter by first briefly reviewing the two distributed, hardware approaches to maintaining coherency, write-invalidate and write-broadcast, and detailing an example protocol from each category. The protocols will be used in section 4.3.3 to illustrate how the values of the sharing metrics vary with different protocols, and again in sections 4.7 through 4.9 in the architecturally detailed simulations. Section 4.3 contains a characterization of sharing and metrics that reflect the characterization. The characterization is the basis for both the sharing analysis and the model of write sharing. Section 4.4 covers additional aspects of the methodology that are particular to the sharing analysis, and section 4.5 presents the sharing results. Sections 4.6 through 4.9 address the applicability of the sharing analysis to the two types of distributed, hardware coherency protocols. First (section 4.6), the model of write sharing is derived from the characterization developed in section 4.3; then, costs reflecting the sources of coherency overhead for each type of protocol are applied to the model to obtain predictions of coherency overhead. Section 4.7 compares the model's results to the architecturally detailed simulations and pinpoints the factors that are responsible for the model's misprediction of the write-invalidate protocols, most importantly, cache block size; sections 4.8 and 4.9 correct the model by incorporating these factors. Lastly, section 4.10 summarizes the results.

4.2. Write-invalidate and Write-broadcast Coherency Protocols

Cache coherency in bus-based, shared memory multiprocessors is usually enforced by one of the distributed, hardware coherency techniques. Under these schemes, when a processor writes to shared data, there are two different procedures that it can follow. It can either invalidate all other cached copies of the data and then update its own without further bus operations. Or, it can broadcast the updates to all other caches, so that all processors always have the most current value of the data. The former method is known as *write-invalidate*, and the latter *write-broadcast*. (For a complete description of the two coherency categories and all protocols in

them, see Chapter 2, sections 2.4.2 and 2.4.3.) I am interested in contrasting the relative performance of these two coherency approaches in copy-back caches. To do this, I shall introduce representative protocols in each category, and then use them in the remainder of the analysis.

Berkeley Ownership [Katz85] is a write-invalidate protocol that has been implemented in the SPUR multiprocessor [Hill86]. It is based on the concept of cache block ownership. A cache obtains exclusive ownership of a block via two invalidating bus transactions. One is a special read operation that invalidates copies of the data in other caches, at the same time it obtains the block for the requesting processor. It is used on cache misses. The second is an invalidation signal that is used on the first cache write hit. Once ownership has been obtained, the cache can update a block locally without initiating additional bus transfers. A block owner also updates main memory on block replacement and provides data to other caches upon request. All cache-to-cache transfers are done in one bus transfer, with no memory update. Because it creates a data writer that can access a shared block without using the bus, we expect Berkeley Ownership to minimize the overhead of maintaining cache coherency in two cases: when there are multiple consecutive writes to a block by a single processor, and when there is little fine-grain sharing.

The *Firefly* protocol uses write-broadcast and has been implemented in the DEC Firefly multiprocessor [Thac88]. Its processors broadcast writes to shared data, but use copy-back for private (non-shared) data. The bus-watching snoops assert a special bus line to indicate sharing, whenever they detect an operation for a block that resides in their respective caches. The scheme has potential performance benefits for both private and actively shared blocks. A processor knows when a block retrieved on a cache miss is private, because the shared line is not asserted. Therefore all subsequent writes to the block can take place without further bus activity. Under Berkeley Ownership, the initial read to shared data provides no hints as to whether the block is actively being shared; therefore the invalidation signal must always be transmitted on the first write to a cached block, even if there are no other cached copies.

Because it broadcasts all shared updates, the Firefly protocol avoids the ping-ponging of shared data among caches that would occur with the invalidations of Berkeley Ownership. However, for data that is shared in a sequential fashion, with each processor completing many accesses to the data before another processor begins, the write-through policy for shared data may degrade bus performance.

Bus-related *coherency overhead* consists of those bus transactions that are (1) required to maintain coherent caches and (2) whose only function is to do so. They are distinct from bus operations that fetch data on cache misses and flush it to memory on block replacements. The two distributed, hardware coherency approaches described above each have different sources of bus-related coherency overhead. In write-invalidate protocols, there are two. The first is the invalidation signal needed to maintain coherent caches. The second is the cache misses that occur when processors need to rereference invalidated data. These misses, called *invalidation misses*, would not have occurred had there been no sharing. They are present because the shared data had previously been written, and therefore invalidated, by another processor.

In the write-broadcast protocols, the coherency overhead stems entirely from the bus broadcasts to shared data. They occur for all updates to data that is contained in more than one cache, and for the first update to an address after the writing processor has the only copy. (In this case the block has been replaced in the other caches.)

4.3. A Characterization of Sharing and the Sharing Metrics

4.3.1. The Characterization

The characterization of sharing serves three purposes. First, it provides an understanding of the memory reference patterns of write shared data, i.e., whether there is sequential or fine-grain sharing. Second, it highlights the essential differences between the protocols, in terms of the bus operations they use to maintain coherency. It therefore explains how the different

patterns of sharing can affect protocol performance. Third, it is used as the basis for sharing models (see sections 4.6 through 4.9) that approximate the coherency overhead of particular protocols.

I have based the characterization on two aspects of memory accessing: the number of sequences of single processor writes to an individual shared address, and the length of these sequences. Both can be portrayed by the notion of a *write run*, which is the central concept of the characterization (Cf. Figure 4-1). A write run is defined as a sequence of write references to a *shared address* by a single processor, uninterrupted by any accesses by other processors. It is

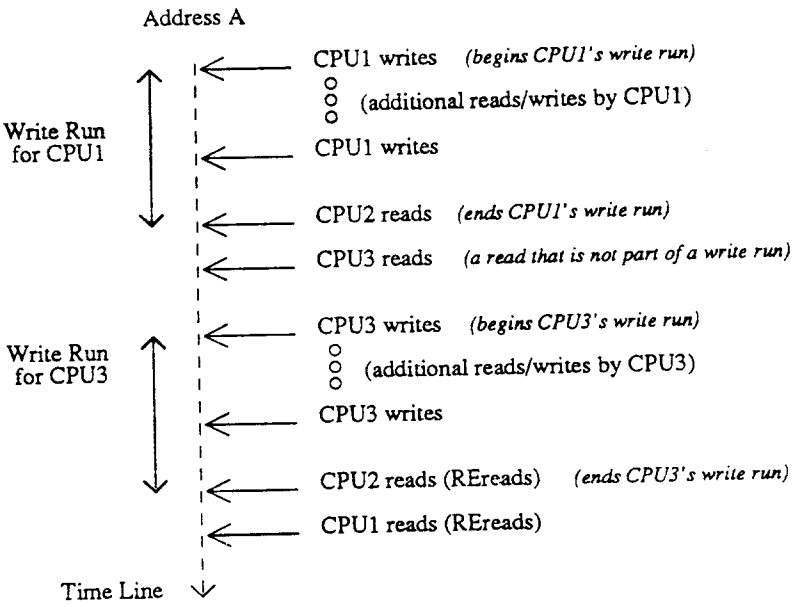

Figure 4-1: Example Write Run for a Shared Address

A write run is a sequence of write references to a shared address by a single processor. It begins with the processor's first write to the address (e.g., the first "CPU1 writes"), and ends with the first access by another processor (e.g., the first "CPU2 reads"). The second occurrence of "CPU2 reads" illustrates an external reread. It is a reread, because the address was read previously (the first "CPU2 reads"), but was invalidated (the first "CPU3 writes"). The vertical arrows denote the time over which the the write run occurs; the number of writes in this interval is the length of the write run (both are at least 2).

initiated by a processor's first write to the address, contains additional reads or writes by that processor, and is terminated by the first access by another processor, either a read or write. (This latter access is called an *external* read or write, because it is external from the point of view of the processor that is the current writer of a write run.) Write runs are nonoverlapping units; and each shared address has a different sequence of write runs.

A write run could contain read references, as well as writes. This analysis will focus on writes (with one exception, external reads, which are discussed below), since shared writes cause coherency overhead, and most reads are handled identically in both protocols. In both the write-invalidate and write-broadcast protocols, additional bus operations are required to maintain coherency on writes. In each case the overhead is different. For example, in Berkeley Ownership the initiation of a new write run results in an invalidating bus operation; however, in the Firefly *each* write in a write run potentially causes a bus operation.

Most reads, on the other hand, do not affect the pattern of shared accesses and consequently the variation in performance due to the particular coherency protocol. The initial read to an address by each processor is always a miss; and, given an infinite cache assumption (explained in section 4.4.1), reads within a write run are all cache hits. *Each type* of read takes the same number of cycles, regardless of the coherency approach adopted.

However, reads are important to track in two cases. First, an external read can be the cause of the termination of a write run. The number of initial, per-processor external reads after a write run is an indication of the number of processors actively sharing an address.[2] If the external reads are *re*reads, they are a cause of coherency overhead in write-invalidate protocols, the invalidation misses described in the previous section. (The two terms, external rereads and

[2] *Passive sharing* is a phenomenon of the write-broadcast protocols. It is caused by shared addresses that were accessed at one time and still remain in a processor's cache. Although they are no longer being referenced by their processor, their presence in the cache drives the shared bus line, causing needless bus broadcasts by the processor that is accessing them. An analysis of the additional broadcasts, over varying cache sizes, and the resulting loss in performance appears in Chapter 6.

invalidation misses, will be used interchangeably, depending on the context. External rereads is a more general, protocol-independent description and describes aspects of the write run characterization and model. Invalidation misses are external rereads, when applied to write-invalidate protocols; I will therefore use this term in discussions of protocol-specific coherency overhead.) Second, including reads in the counts of references within a write run provides an accurate basis for measures of locality.

The write run characterization of sharing portrays the sources of bus-related coherency overhead for the write-invalidate and write-broadcast protocols. For write-invalidate, coherency costs occur for the first write in a run, which is the invalidation signal, and for the external rereads or invalidation misses. For write-broadcast, each write within a write run causes coherency overhead.

[Agar88] describes a similar characterization, based on the notion of *pings* and *clings*. A ping is an external read or write; a cling is one of the references within a write run. Their paper focuses on the temporal locality of shared references by measuring the time distribution between pings and clings. They found that the temporal locality of pings and clings was equivalent, and hypothesized that the fine granularity of parallelism in their programs (roughly 100 instructions) was responsible for the low ping locality figures. Other portions of this paper tie into the write run metric results from the analysis of sharing; the two sets of results will be compared in section 4.5.

4.3.2. The Write Run Metrics

The write run metrics used to analyze sharing appear in Table 4-1. The length of a write run is measured in numbers of writes. Beginning with the first write to a shared address by a particular CPU, the number of writes by that CPU is counted until the first access by another CPU. The count of per-processor first external rereads is the number of *different* processors that reread the address after a write run. This metric indicates the number of processors that are

actively sharing the address.

The sharing ratio and the number of busywaiters measure the level of sharing and contention, respectively. The sharing ratio is the total number of write runs divided by the number of shared addresses written by one or more processors. The sharing ratio for one address is simply the number of write runs that occurred for that address. The metric of interest here is the average over all write-shared addresses. (It is not necessary to normalize the sharing ratio to the length of the trace, since all trace snapshots are identical, approximately 300,000 references per processor.) The sharing ratio provides intuition about the level of sharing. Lower values indicate less sharing, while higher values signify more. The metric is an average for the entire trace, covering accesses to both locks and applications shared data. The number of busywaiters, on the other hand, measures contention for a subset of the addresses, i.e., only those for locks. The number of busywaiters is the number of processors that are blocked waiting for a lock when it is unlocked. Like the sharing ratio, a higher metric value indicates greater lock contention.

4.3.3. Applying the Metrics

The write run metrics are useful for analyzing the performance tradeoff between the write-invalidate and write-broadcast protocols. A long write run suggests that a write-invalidate

Metric	Aspect of Sharing It Measures
Count of writes in a write run	consecutive single processor usage for shared data
Count of per processor first external rereads	sharing
Sharing ratio	sharing
Number of busywaiters for a lock	contention

Table 4-1: Sharing Metrics Based on Write Runs

This table lists the write run metrics and the aspects of sharing that they measure. The metrics are described in detail in the text.

protocol should be adopted. After the invalidation signal for the first update, all other writes by that processor can take place locally; therefore the cost of the invalidation signal is amortized over the entire run. On the other hand, write-broadcast would perform better with short write runs, particularly those of length one. In this case both approaches incur a coherency-related bus operation for the first write; but write-broadcast avoids the read misses of the write-invalidate schemes.

The number of external rereads measures a source of coherency overhead in the write-invalidate protocols only. A large number of external rereads indicates that the addresses would have been needlessly invalidated had the coherency protocol been write-invalidate, and that a write-broadcast scheme would therefore have been preferred. On the other hand, a low number of rereads suggests that the invalidations may have done little harm.

The performance of the two coherency approaches depends on the combined effects of these measures. Even if the write run is short, but there is no sharing, e.g., no external rereads for the address, a write-invalidate scheme still might produce the better performance. The opposite situation calls for write-broadcast: if the number of bytes transferred by external rereads is greater than those updated by the number of broadcasts in the write run length, that approach is better.

Write run length may also indicate whether there is contention or fine-grain sharing for a shared address, or whether it is shared sequentially by each processor over long periods of time. Short write runs, particularly those occurring in a short time interval, suggest that a processor's algorithmic use of the data was interrupted by other CPUs also referencing the address, i.e., fine-grain sharing. Contention is also greater: the greater the number of external rereads, the higher the sharing ratio and the larger the number of busywaiters for a lock. As was explained in section 4.2, write-broadcast protocols are well suited for periods of contention, while write-invalidate performance suffers.

4.4. Sharing Analysis Criteria

The objective of the sharing analysis is to focus on the sharing inherent in the application programs, and abstract away the architectural and implementation details of the multiprocessor, which could affect the pattern of sharing. For example, in write-invalidate protocols the unit of invalidation is the cache block. If the block size is larger than a word, then invalidations due to processor writes will unnecessarily nullify the other words in the block. If those words are subsequently accessed by different processors, additional bus reads will be incurred to obtain the data. The number, type and order of these bus operations will depend on the particular block size chosen. If the sharing analysis focuses on the shared addresses being referenced rather than the block size, the results will not be perturbed by changes in block size.

An analysis of sharing that is independent of the underlying architecture and the coherency protocol has several advantages. First, it provides an understanding of the memory reference pattern of write-shared data that is inherent in the applications themselves. These results can be used to determine coherency overhead for a variety of coherency protocols and cache architectures. Second, for a single trace, only one simulation need be done (as opposed to one for each combination of architecture and protocol parameter values). Third, the sharing simulator is simpler to implement than one that precisely models the features of a particular architecture and coherency protocol. (For example, the details of bus arbitration and bus transactions, snoop activity, and cache controller/snoop interaction over the use of the cache were omitted in the sharing analysis). Therefore both simulator design time and simulation run time are shorter. For these reasons, the sharing study was conducted as independent of the architecture, implementation, and coherency protocol as possible. The next two subsections describe how this was accomplished.

4.4.1. Architecture/Implementation Independence

Independence from the underlying multiprocessor architecture and its implementation is achieved in several ways. First, the simulations assume *infinite caches* to eliminate the effect of cache size on block placement. In an infinite cache there is room for all references, and no blocks need to be evicted and consequently reaccessed. Reaccessing increases both bus traffic and miss ratios of shared data directly, and has an indirect effect by altering the order of processor accesses to the bus, thereby changing the pattern of shared accesses.

Second, addresses are tracked, rather than cache blocks, so the analysis is based on the unit of access of the CPU. This eliminates the effect of changing the cache block size, and is equivalent to setting the block size to one word in the simulator.

Third, all memory references take the same amount of time, regardless of whether they are reads or writes, hits or misses, or the misses are satisfied by main memory or another cache. The differences in the amount of time required to carry out these alternatives (in a real system) is sensitive to the memory organization, particularly memory latency, the bus transfer time, and the cache controller implementation.

Fourth, memory references are satisfied on a per processor, round robin basis, to give all processors equal processing time.

Lastly, the cycle time per instruction is a constant. Varying the instruction time to mirror the underlying implementation affects instruction latency, which, again, alters the global sequence of shared accesses by modifying the order in which processors obtain the bus. An argument could be made that instruction cycle times of the generation machines should be included in the simulation, because the particular choice of instructions reflects the semantics of the parallel algorithm. However, in the current programming paradigm (explained in Chapter 3, section 3), all parallel processes are executing the same code. Thus the variation in instruction times would be identical across processors.

4.4.2. Coherency Protocol Independence

Results of the architecturally detailed simulations (see section 4.7) indicate that the metrics associated with multiprocessor performance, and the sharing aspects in particular, are sensitive to the timing differences introduced by the choice of cache coherency protocol. The differences affect the amount and pattern of sharing and the execution time of the program in three ways: directly, by causing different bus events to occur, and indirectly by (1) altering the multiprocessor (systemwide) order of references to shared data and (2) varying the amount of busywaiting needed to obtain a lock. Therefore the sharing simulations were done without introducing protocol-related variations, i.e., with no bus-related overhead involved in carrying out the sharing operations. Under this coherency model, accesses to shared data are still tracked and coherency maintained, but with no cost in time.

4.4.3. Synchronization

Two aspects of processor synchronization (and their corresponding overhead) are still included: barriers and busywaiting for locks. Both of these constructs are reflections of the underlying algorithm. Barriers prevent processes from executing beyond a certain point in the algorithm, until all parallel processes have reached that point. They are used to guarantee a correct ordering of phases of the program, e.g., to separate time steps in a circuit simulation. Busywaiting is more difficult to justify. One could argue that busywaiting should be eliminated, because it reflects the timing constraints of the underlying architecture and the policy of the cache coherency protocol,[3] as well as the algorithm. However, under the assumption of architecture and coherency protocol independence, the busywaiting that occurs is a reflection of contention for the shared locks inherent in the application's flow of control.

---

[3] For example, the extent to which busywaiting is done either over the bus or locally in the cache varies among protocols.

4.5. Results of the Sharing Analysis

This section contains the results from the architecture- and protocol-independent sharing analysis. Statistics for the traces by type of reference appear in Table 4-2. The nonsharing-related figures are within the normal range of uniprocessor program behavior. The important figure for sharing analyses is the low percentage of shared accesses[4], particularly to write-shared data. Recall that the traces contain applications references only; no references to shared operating systems data structures are included. Therefore, unless operating systems activity adds substantially to the number of write-shared references,[5] memory references due to coherency overhead will be a small component of the total. However, they may still comprise a substantial proportion of total bus operations, since most references to write-shared data result in a bus transaction, and many board-level caches have fairly low miss ratios (see Chapters 5 and 6).

A further classification of shared references by type of data appears in Table 4-3. Note the preponderance of references to applications shared data level over the locks that protect it. In three of the programs applications shared data is only accessed within critical sections. The paucity of references to locks suggests that there is little contention for this data. A higher percentage of reads over writes for lock data (e.g., in CELL and SPICE) means that there was busywaiting for the lock. A lock write value exactly twice that of the reads (VERIFY) signifies a total absence of busywaiting. The locking algorithm is the test-and-test-and-set sequence used in the SPUR multiprocessor: the read is the initial access of the lock; the two writes are for setting and clearing. (TOPOPT does not use locks; it protects its shared data with barriers and the semantics of the algorithm, i.e., within a particular phase of the program there are multiple readers for a shared address, but only one writer.)

---

[4] Shared data is defined to be those addresses that reside in a program's shared memory. References to them are included in these figures, whether the data is currently being shared or not.

[5] Results from the first trace-driven study that includes operating systems references indicate otherwise. [Agar88] found that references to shared data in MACH comprised from .5 to 1.8 percent of total references and 3.5 to 12.5 percent of total shared data references (for both user and system).

Basic Trace Statistics									
Trace	Refs (1000s)	Code	Data	Reads (Data)	Writes (Data)	Private (Data)	Shared (Data)	Read Shared (Data)	Write Shared (Data)
		(proportion of total references)							
CELL	3,732	.546	.454	.356	.098	.310	.144	.131	.013
SPICE	1,538	.629	.371	.256	.115	.283	.089	.070	.019
TOPOPT	3,300	.662	.338	.316	.022	.196	.142	.139	.003
VERIFY	3,605	.682	.318	.255	.063	.187	.131	.121	.010

Table 4-2: Basic Trace Statistics

Basic Trace Statistics: Details of the Shared Data								
Trace	Shared Refs (1000s)	Applications Shared Data			Locks			Shared Data Space (Kbytes)
		Total	Reads	Writes	Total	Reads	Writes	
		(proportion of shared references)						
CELL	537	.915	.827	.088	.085	.081	.004	326.6
SPICE	136	.899	.697	.202	.101	.091	.010	26,431
TOPOPT	470	1.000	.980	.020	.000	.000	.000	22.2
VERIFY	472	.993	.919	.074	.007	.002	.005	114.5

Table 4-3: Shared Data Trace Statistics

The number of references is the total processed in the sharing simulation. The proportions are the arithmetic means across all processors. They were calculated from the simulations results, assuming architecture and protocol independence. The nonsharing-related figures are within the normal range of uniprocessor program behavior. The proportion of reads to writes only seems high, because the ratios include references to shared data. Ratios of reads to writes for private data are comparable to other studies (for example, [Smit85]), for three of the traces (2.6 for CELL, 1.9 for SPICE and 2.6 for VERIFY). The only exception is TOPOPT (9.1). The shared data ratios are in line with previously published figures. In [Dare87] the proportion of shared accesses to total was .03, .14 and .12 for three scientific applications (molecular dynamics, Fast Fourier Transform and fluid dynamics, respectively). Ratios of shared to total data were .24 for SPICE, .32 for CELL, and .42 for TOPOPT and VERIFY. The figures for SPICE and CELL agree with the .27 average for three similar applications reported in [Agar88]. The higher proportion of private to shared data for SPICE is probably attributable to accesses to local copies of read-shared data. The column headed "Shared Data Space" is the number of bytes allocated to all shared data. In all traces except SPICE, it is the amount of shared memory required to execute the program on the particular input used. SPICE was written in Fortran; therefore the shared space was statically allocated to fit inputs of varying sizes.

Histograms for the length of the write runs and the number of external rereads are shown

Write Run Length Histogram								
Run Length Bins	Traces							
	CELL		SPICE		TOPOPT		VERIFY	
	% Write Runs	% Writes	% Write Runs	% Writes	% Write Runs	% Writes	% Write Runs	% Writes
1	71.6	26.3	66.1	35.2	60.5	9.2	30.4	3.8
2	13.9	10.2	21.5	22.9	10.9	3.3	43.7	11.0
3	2.8	3.1	3.6	5.7	4.6	2.1	5.4	2.0
4	2.1	3.1	4.0	8.6	7.3	4.4	4.7	2.4
5	0.9	1.7	0.7	2.0	3.1	2.3	1.4	0.9
6	0.4	1.0	0.1	0.2	1.1	1.0	1.8	1.4
7	1.1	2.8	0.4	1.4	1.1	1.2	0.7	0.6
8	0.7	1.9	0.4	1.7	1.6	1.9	0.8	0.8
9	0.9	2.9	0.0	0.0	0.5	0.7	0.2	0.3
10	1.0	3.7	0.0	0.2	0.2	0.3	0.4	0.5
11	1.1	4.5	2.6	14.9	0.4	0.7	0.2	0.3
12	0.3	1.5	0.4	2.8	0.1	0.2	0.7	1.1
13	0.4	1.9	0.0	0.1	0.1	0.2	0.5	0.8
14	0.3	1.4	0.0	0.0	0.1	0.1	0.4	0.6
15	0.0	0.0	0.0	0.0	0.4	0.9	0.5	1.0
16	0.1	0.3	0.0	0.1	0.1	0.3	0.1	0.3
17	0.1	0.4	0.0	0.0	0.2	0.4	0.2	0.4
18	0.1	0.9	0.0	0.0	0.2	0.6	0.2	0.3
19	0.1	1.0	0.0	0.0	0.4	1.2	0.0	0.1
20	0.3	2.5	0.0	0.0	0.3	0.8	0.1	0.3
>20	1.7	15.6	0.2	2.3	6.8	46.2	7.6	51.2
Total Write Runs	20959		15684		1864		5834	
Avg. Write Run Length	2.36		1.84		5.13		6.37	
Total Writes		57063		29439		12231		46473

Table 4-4: Length of the Write Runs

This histogram depicts the percentage of write runs that have a particular write run length and the percentage of total writes that they contain. The traces were heavily biased toward write runs that contained only one write. With the exception of VERIFY, approximately two-thirds of the write runs for each trace had one write. Despite this, the average write run length for TOPOPT was long enough to suggest that a write-invalidate coherency protocol would be most appropriate for it.

in Tables 4-4 and 4-5. For two of the traces, CELL and SPICE, the write runs are short. Their

average write run lengths are 2.36 and 1.84 writes, respectively, and most of their write runs contain only one write (.72 for CELL; .66 for SPICE). In isolation, write runs this short argue for a write-broadcast protocol. However, the situation *may* change given the few external rereads. The average number of external rereads for CELL and SPICE is close to one (CELL = 1.26, SPICE = .89), and a high percentage of their write runs were terminated by one or fewer rereads (CELL = 79%, SPICE = 99.5%).[6] With the number of rereads this small, the invalidations in a write-invalidate scheme would cause little additional coherency overhead in terms of invalidation miss bus traffic. However, both write run length and the number of rereads are

External Rereads Histogram				
External Rereads Bins	Traces			
	CELL %	SPICE %	TOPOPT %	VERIFY %
0	29.7	11.4	68.4	12.1
1	49.6	88.1	9.3	76.1
2	10.8	0.4	8.1	8.2
3	3.3	0.1	2.7	3.6
4	1.4	0.1	0.5	0.0
5	0.8	0.0	0.2	0.0
6	1.1	0.0	0.2	0.0
7	0.9	0.0	0.0	0.0
8	1.3	0.0	0.0	0.0
9	0.8	0.0	0.0	0.0
10	0.5	0.0	10.6	0.0
11	0.1	0.0	0.0	0.0
12	0.0	0.0	0.0	0.0
Total External Rereads	17506	9575	886	2961
Avg. External Rereads	1.26	0.89	1.44	1.03

Table 4-5: Number of External Rereads Following a Write Run

This histogram depicts the percentage of write runs that were followed by a particular number of external rereads. The graph indicates that a single or no external rereads terminated most of the write runs in all traces. Tnis means that there was a very low level of multiple processor contention for the shared variables, and argues for a write-invalidate coherency protocol.

[6] No external rereads occur when the end of a write run is the beginning of the next. In this case the terminating access is a write by the new processor.

sufficiently low that a clear protocol choice cannot be made.

The average write run lengths for TOPOPT and VERIFY are longer (TOPOPT = 5.13, VERIFY = 6.37); and for VERIFY the write runs of length one constitute less than one third of the total. The average number of external rereads are as low as for the other two traces (TOPOPT = 1.44, VERIFY = 1.03), and the percentage of write runs ending with one or fewer rereads is also comparable (TOPOPT = 78%, VERIFY = 88%). The combination of the long write run length and the low number of external rereads for these traces indicates a match with a write-invalidate protocol.

Results in [Agar88] are lower than the write run length figures reported here, presumably because of the finer granularity of parallelism in the programs. The average write run length for data in their three CAD applications is 1.5. Including operating systems references only increased the average write run length by 5 percent.

Three factors indicate that contention in the traces is low. The first is the low number of external rereads mentioned above. Few rereads indicates that few processors are simultaneously accessing the same shared data. Second, the sharing ratio is also low (see Table 4-6). A low level of sharing indicates little contention. The ratio of write runs per total shared write addresses referenced averages 2.7 during trace runs of 300,000 memory references per processor. The amount of computation in the applications is large relative to the frequency of write-shared references, and, as established above, the pattern of access to these addresses is sequential. Therefore a 300,000 reference snapshot is not sufficient to capture repeated sharing of the data in the work queue that occurs over many iterations of the algorithm. This is a comment on the low level of sharing in the traces, rather than the insufficient size of the trace sample. If a larger section of trace were analyzed, the same level of sharing would have been exhibited: both the sharing ratio and the time period (measured in numbers of memory references) would

Sharing Ratio		
Trace	Value	Time Period (mem refs in 000's)
CELL	4.7	3,732
SPICE	2.5	1,538
TOPOPT	1.8	3,300
VERIFY	1.9	3,604

Table 4-6: Sharing Ratio

The level of write sharing in the traces is reasonably low. This is indicated, in part, by the low number of write runs per shared write address (the sharing ratio) over a long period of time. Time is measured by the total number of memory references processed. Total write-shared addresses is a dynamic measure of those locations accessed during the period.

Busywaiters Histogram			
Bin	Traces		
	CELL (%)	SPICE (%)	VERIFY (%)
0	86.6	81.8	99.3
1	9.7	14.9	0.7
2	2.7	2.4	0.0
3	0.8	0.6	0.0
4	0.2	0.3	0.0
Total Busywaiters	998	636	1097

Table 4-7: Number of Busywaiters

This histogram depicts the number times a processor was blocked from a critical section because another processor was executing in it. The snapshot was taken when the lock was unlocked, and the count is of the number of processors busywaiting for it. The figures indicate that there was almost no contention for the locks. At the very least almost 82 percent of all locks were unlocked with no other processor waiting. (TOPOPT is not depicted, because it does no locking.)

increase.[7]

Third, there are few processors busywaiting for locks when the lock is unlocked (see Table 4-7). Between 82 and 99 percent of unlocks occurred with no other processor wanting the lock. This behavior is partially determined by the programming paradigm. When the

[7] As mentioned in Chapter 3.5.3, one and two million reference snapshots of SPICE were also simulated, and the ratio of the sharing ratio to total memory references was almost constant across the three snapshot sizes.

programs first begin execution, there is unusual contention for the locks protecting the queue of work, since all child processes try to take their first unit of work simultaneously. However, only one process will obtain access to the queue at a time. Since each process does a comparable amount of computation, they will thereafter access the queue in the same order, spaced in time by the execution time of the critical section. This self-scheduling is disrupted by synchronization barriers, which are used to separate phases in the computation. The disruption causes more busywaiting and therefore an increase in references to the locks. However, it occurs infrequently, particularly when compared to the longer periods of self-scheduling.

Given all three factors (the low number of external rereads, the low sharing ratios and little busywaiting), I conclude that the short write runs depicted in Table 4-5 result from the processors' intention to write to the shared addresses only once, rather than the write sequences being interrupted by accesses from other processors.

In summary, the sharing results of the simulations, *independent of the architecture and coherency protocol*, indicate sharing behavior more appropriate for write-invalidate protocols than write-broadcast for the traces examined. Write-invalidate's performance advantages stem primarily from the lack of contention for shared data (the small number of external rereads), and for two of the traces, the long length of the write run. TOPOPT and VERIFY had average write run lengths of 5.1 and 6.4, respectively, which allows a generous margin between the cost of one invalidation signal and a per-write bus broadcast within each write run. Of all the traces, SPICE and CELL were the best candidates for a write-broadcast protocol, because of the combination of shorter write runs, slightly longer busywaiting sequences, and a higher sharing ratio.

4.6. The Write Run Model

Architectural modeling is a useful performance technique for several reasons. First, it provides good intuition about the factors that affect computer performance. It reduces the com-

plexity of the architecture to a few key components, and thus provides a means to easily analyze the interactions among them. The purpose of the write run model is to evaluate coherency overhead; therefore it is restricted to coherency-related bus events. The model captures references to shared data that either degrade performance by causing additional bus traffic or are handled differently in the different protocols. It portrays write sharing activity only, because this is where the write-invalidate and write-broadcast protocols differ. It is assumed that both approaches have similar uniprocessor bus utilization, i.e., for private data and instructions. In addition, both are copy-back schemes. Therefore the model does not include these activities.

A second advantage of modeling is that it produces results relatively quickly. By changing the model's parameter values, one can explore a large design space in a much shorter period of time than with trace-driven simulation. This convenience is especially important for multiprocessor studies, because the simulation time is proportional to the number of processors being simulated. For the studies in this dissertation, each architecture and protocol simulation required 7.25 hours of simulation time on an unloaded VAX 8600.

The simple model of write sharing developed here is based on the write run characterization. In the model, each state represents a different write run activity. A shared address is assigned to a state, based on its past write activity and current reference. When a shared address is accessed, a state transition occurs, as illustrated in the state diagram in Figure 4-2. A transition is made to the Different Write Run state by a write to a shared address by a CPU other than the current writer. The number of transitions is the count of write runs for the address. The Same Write Run state is entered each time a writer continues to write to the address. The number of transitions here is the total of all write run lengths for that address, excluding the first write in each run. The transition to End of Write Run is made by the first external read to the address by each CPU. The total is the sum of all such reads.

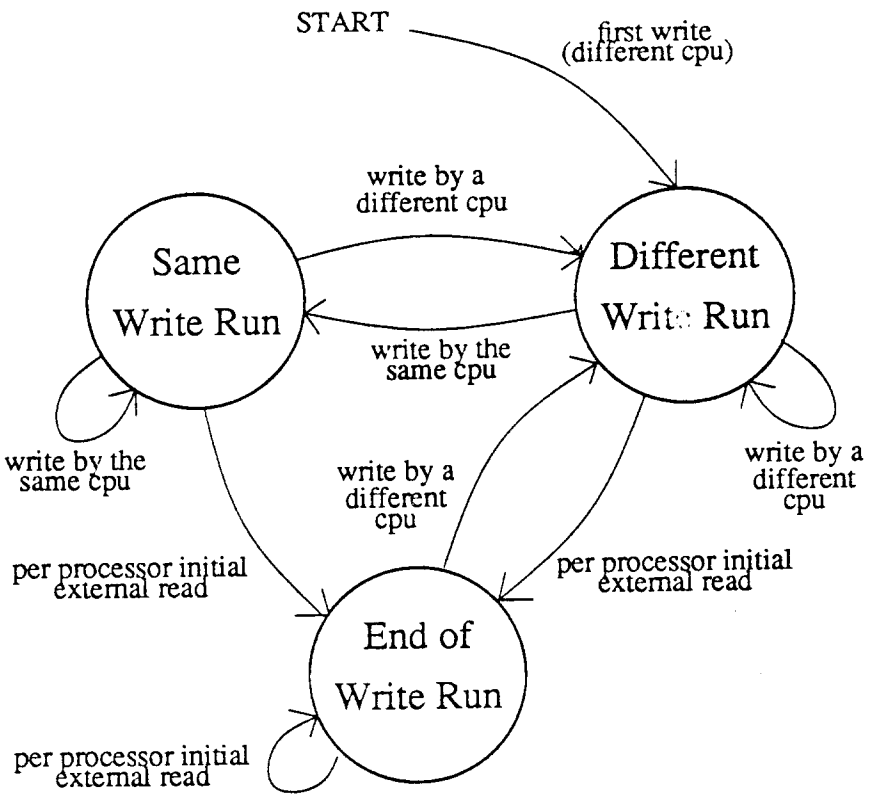

Figure 4-2: Model of Sharing Based on Write Runs

This finite state diagram reflects the model of sharing developed from the characterization based on write runs. A transition is made to the Different Write Run state by a write to a shared address by a CPU other than the current writer. The Same Write Run state is entered each time a writer continues to write to that address. The transition to End of Write Run is made by the first external read to the address by each CPU. By assigning a coherency protocol-dependent cost to each arc in the state machine, an approximate cost of sharing for a particular cache coherency protocol can be determined.

The model can be used to quantify the intuitive conclusions of the last section. Relative coherency protocol performance is determined by assigning costs (in cycles) to each arc in the finite state diagram and multiplying by the transitions for each arc, summed across all shared addresses. The costs are a measure of the overhead of sharing traffic for a particular cache coherency protocol and are based on the timing constraints in the implementation of the SPUR multiprocessor. Cost values assume that the program is in *sharing steady state*. In sharing

steady state shared data is actively being shared by multiple processors. Its use in simulations ensures that statistics are not gathered during sharing startup, during which only one processor has cached the shared data and some coherency costs are not incurred. (Sharing steady state is a notion similar to cache steady state. Cache steady state also avoids a startup situation, i.e., initial cache filling, so that cold start misses are not included in cache miss ratio calculations.) Sharing steady state implies that (1) a shared access is not the first reference to that address by a processor, i.e., after the access, at least two caches have copies; and (2) each external read is actually a reread, i.e., the processor has previously accessed the address. The first stipulation insures that the number of bus updates in a write-broadcast protocol can be approximated by the write run length (the number of transitions to Different Write Run state plus those to Same Write Run state); the second that external rereads in write-invalidate protocols are equivalent to external reads (the number of transitions to End of Write Run). The arc costs for Berkeley Ownership and the Firefly are depicted in Table 4-8.

Costs of Transitions in the Write Run Sharing Model				
Arc	Berkeley Ownership		Firefly	
	Bus Operation	Cost (cycles)	Bus Operation	Cost (cycles)
Write by a Different CPU	invalidation signal	12	word transfer	11
Write by the Same CPU	no cost	0	word transfer	11
First Per-processor External Rereads	block transfer	18	no cost	0

Table 4-8: Costs of Transitions for Berkeley Ownership and Firefly

This table classifies coherency overhead by type of bus operation for Berkeley Ownership and the Firefly. For each state transition in the state diagram (Figure 4.2), the bus operation required and its costs in cycles are depicted. All bus operations include cycles for address translation, bus arbitration, the bus operation and the appropriate snoop response, and snoop/cache controller interaction over updating the cache controller's copy of the state. The block is assumed to be eight words. The small time difference between the invalidation signal and a one-word transfer is caused by the update of both copies (snoop and cache controller) of the state for the former. The exact choice of cycle value is based on the implementation of the SPUR multiprocessor.

Figures 4-3 and 4-4 illustrate the model when applied to the write-invalidiate and write-broadcast protocols. In each diagram, state transitions are depicted only for valid coherency bus operations. The coherency cost to Berkeley Ownership is the invalidation signal for the first write in a write run and the rereads for data that were invalidated. Total coherency cost is based on the sum of these bus operations. The coherency overhead of the Firefly protocol is the sum of all broadcast writes to shared data. As stated above, this can be approximated by the total length of all write runs.

The model's position in the total methodological sequence is depicted in Figure 4-5. The parallel traces are input to a multiprocessor simulator, which is either architecturally abstract

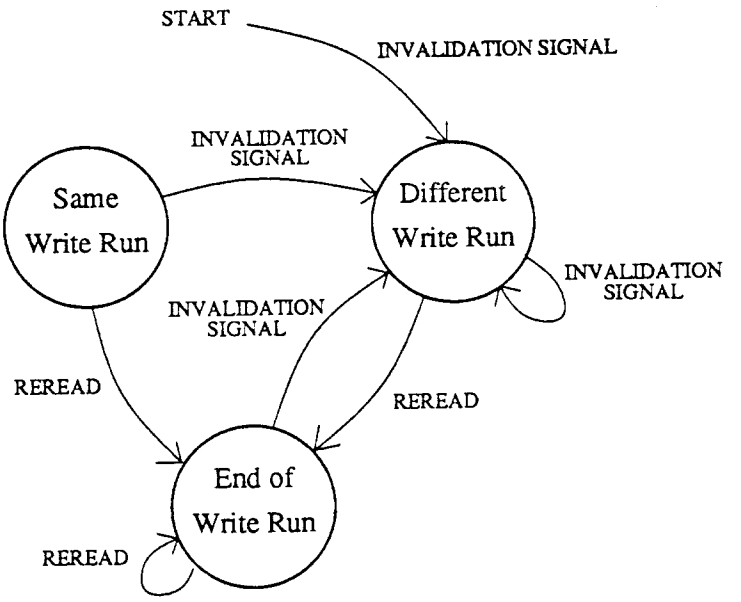

Figure 4-3: Write Run Sharing Model for Berkeley Ownership

This state machine diagram applies the write run model to the write-invalidate protocols, as represented by Berkeley Ownership. State transitions are only depicted for valid coherency bus operations. For write-invalidate they are invalidation signals to the Different Write Run state and external rereads to the End of Write Run state.

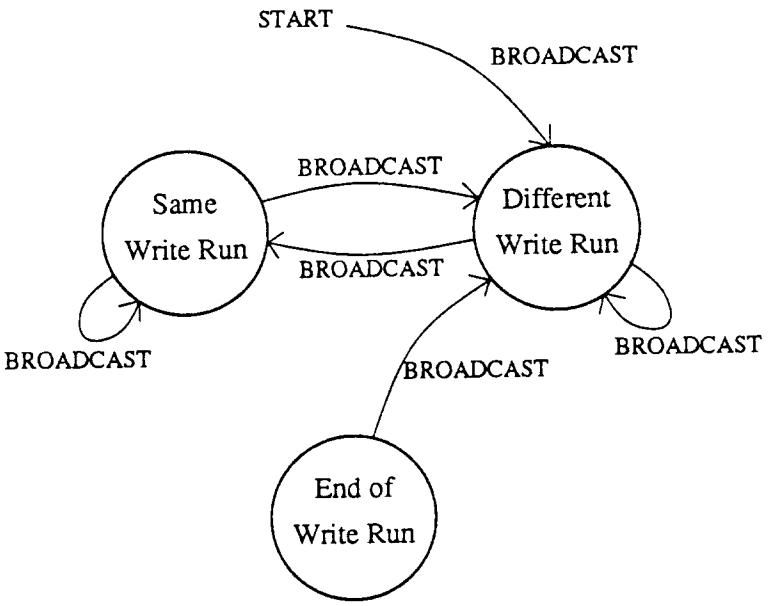

Figure 4-4: Write Run Sharing Model for the Firefly

This state machine diagram applies the write run model to the write-broadcast protocols, as represented by the Firefly. State transitions are only depicted for valid coherency bus operations. For write-broadcast they are bus updates to write-shared data.

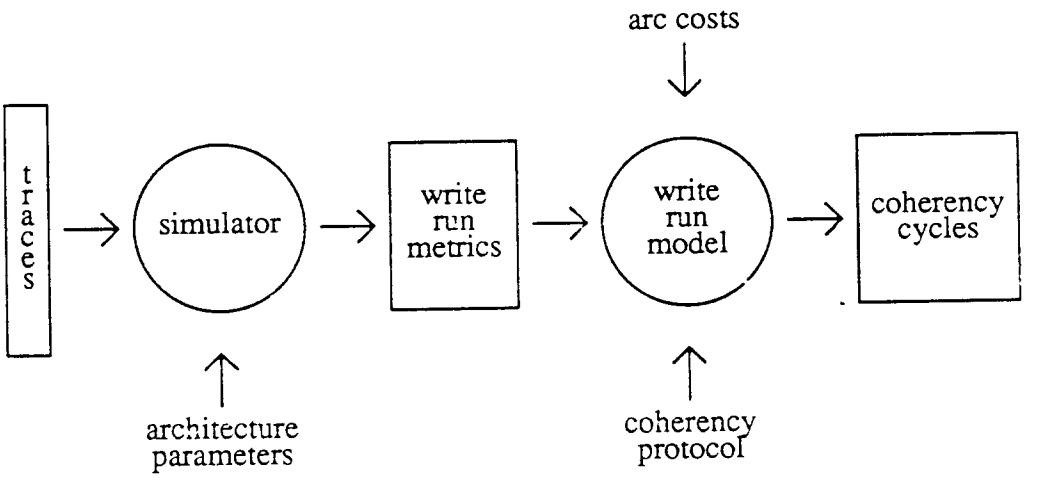

Figure 4-5: Methodology

This diagram depicts the trace methodology used for the sharing analysis and modeling, both based on the write run characterization. The details are explained in the text.

(for the sharing analysis) or very architecturally detailed (for the realistic simulations). The simulator's output are the write run metrics that were presented in section 4.5 (write run length, external rereads, sharing ratio and number of busywaiters), and other figures which will comprise the results in Chapters 5 and 6. The write run metrics from the sharing analysis provide the transition frequency inputs to the write run model; other inputs to the model are the protocol-specific arc costs and the protocol being modeled. The output produced by the model is total coherency overhead, measured in bus cycles.

The results of the protocol comparison appear in Table 4-9. The absolute values of the cycle counts should not be taken literally, because of the architecture- and protocol-independent nature of the studies. What is important is the relative performance of the protocols for a particular trace. The figures of Table 4-9 support the conclusions of the last section, given SPUR cost assignments; namely, that for TOPOPT and VERIFY, write-invalidate protocols (as represented

Cost of Berkeley Ownership & Firefly in the Write Run Sharing Model						
Trace	Coherency Protocol	Diff. Write Run (arcs)	Same Write Run (arcs)	End of Write Run (arcs)	Coherency Overhead (cycles)	Normalized to Berkeley Ownership
CELL	Berk. Own.	20959	28460	22060	648588	1.00
	Firefly				543609	0.84
SPICE	Berk. Own.	15684	13167	8558	342252	1.00
	Firefly				317361	0.93
TOPOPT	**Berk. Own.**	1864	7700	1276	45336	1.00
	Firefly				105204	2.32
VERIFY	**Berk. Own.**	5834	31336	3064	125160	1.00
	Firefly				408870	3.27

Table 4-9: Write Run Model Comparison of Berkeley Ownership & Firefly

This table depicts the number of state transitions for each state in the write run model. The total number of cycles is obtained by multiplying the cost of each arc transition times the arc costs in Table 4-8. The bold entries indicate which of the protocols had better performance for the particular trace, according to the architecture-independent write run analysis of sharing.

by Berkeley Ownership) obtained significantly better performance than the write-broadcast protocols (as represented by the Firefly). And, conversely, CELL and SPICE have less coherency overhead with write-broadcast, although the performance advantage is smaller.

It should be pointed out that coherency cycles are sensitive to the overhead in the bus operations and the transfer size on a block read. In the SPUR implementation, both are high. For example, the cache controller was implemented assuming that the priority for using the cache belonged to the processor rather than the snoop. Therefore all arc costs include cycles for the snoop's negotiating to obtain use of the cache, and acknowledging that it has finished. In addition, the block transfer cost is based on an eight-word block size. If the arc costs had reflected a more optimized implementation, e.g., that used in the Firefly multiprocessor, the cycle cost would have been much lower.[8]

4.7. Architecture-Dependent Simulations of Snooping Protocols

I determined the accuracy of the write run model in evaluating the performance of coherency protocols in a real system by comparing the model's predictions with simulation results, using realistic architecture and protocol parameters. The architecture-independent assumptions of the sharing analysis (described in section 4.4.1) were dropped in favor of more specific premises: (1) the realistic simulations tracked the entire coherency block instead of the one-word unit of CPU access; (2) SPUR's 128K byte, direct mapped cache replaced the infinite cache, and its 32 byte block served as the coherency block; (3) cycle times for bus operations were based on the SPUR implementation, rather than being a constant; (4) specific coherency protocols, Berkeley Ownership and Firefly,[9] were implemented, and appropriate cycles reflecting the costs of their coherency operations replaced the assumption that cache coherency was free; and, (5) in general, the CPU, memory system and bus architecture closely matched

[8] The comparable figures for an implementation similar to the Firefly multiprocessor (using MicroVax II's) would be 4 cycles for a word transfer, and, presumably, 4 for an invalidation and 11 for a block transfer, assuming the SPUR block size.

that of SPUR.

The results of the simulations appear in Table 4-10. The data is the number of bus operations used to maintain coherency in sharing steady state, and the cycles required to carry them out. Again, the coherency cost (in bus operations) with Berkeley Ownership are the invalidation signals and the reaccesses of invalidated data (corresponding to the sum of Different Write Run and End of Write Run figures in Table 4-9); for the Firefly, it is the total number of write-broadcasts to shared data (the sum of Different Write Run and Same Write Run).

The results indicate that the write-run model was a good predictor of coherency overhead, but for the write-broadcast protocols only (see Table 4-11). The percentage difference between

Cost of Berkeley Ownership & Firefly in Realistic Simulations						
Trace	Coherency Protocol	Diff. Write Run [Inval. Signal] (arcs)	End of Write Run [Inval. Misses] (arcs)	Diff. + Same Write Run [Write Bdcasts.] (arcs)	Coherency Analysis (cycles)	Normalized to Berkeley Ownership
CELL	**Berk. Own.**	10062	12507		275106	1.00
	Firefly			49419	543609	1.96
SPICE	**Berk. Own.**	4643	3959		132954	1.00
	Firefly			28937	318307	2.39
TOPOPT	Berk. Own.	7222	5728		225702	1.00
	Firefly			9564	105204	.47
VERIFY	Berk. Own.	16400	25345		629628	1.00
	Firefly			37156	408716	.65

Table 4-10: Comparison of Berkeley Ownership & Firefly in Realistic Simulations

The table contains the number of bus operations needed to maintain cache coherency, assuming a SPUR-like multiprocessor and sharing steady state, and using either Berkeley Ownership or the Firefly protocols. The results indicate relative coherency performance opposite to what was predicted by the write run model. Berkeley Ownership produced fewer coherency cycles than the Firefly for CELL and SPICE, but had more overhead for TOPOPT and VERIFY. (The bold protocol names indicate the protocol with the lower coherency overhead.)

[9] The implementation for the Firefly protocol includes the shared bus line.

Comparison of Realistic Simulation to Models		
Trace	Coherency Protocol	Architecture Independent Model (percent)
CELL	**Berkeley Ownership**	-135.76
	Firefly	0.00
SPICE	**Berkeley Ownership**	-155.42
	Firefly	0.30
TOPOPT	Berkeley Ownership	79.91
	Firefly	0.00
VERIFY	Berkeley Ownership	80.12
	Firefly	-0.04

Table 4-11: Comparison of Write Run Model to Realistic Simulation

This table contains the percentage difference in total coherency cycles between the realistic simulations and the write run model, using the actual cycles as the base. For the Firefly, the total number of cycles required to carry out the operations matches those approximated by the model. However, for Berkeley Ownership, in which coherency operations take place on an entire cache block rather than a word, the effects of the cache block size outweigh those of the sharing pattern in the application. (The bold protocol names indicate the protocol with the lower coherency overhead.)

the model's predictions and the actual Firefly cost was negligible for all traces. However, the model mispredicted coherency overhead for write-invalidate. The cycle discrepancy manifested itself in absolute amounts and in relative protocol performance. The absolute disparity ranged from 2.4 (CELL) to 5.0 (VERIFY) times. For CELL and SPICE the model forecasted a performance loss for Berkeley Ownership relative to the Firefly (19.3 percent more coherency cycles for CELL, 7.8 percent more for SPICE); and a savings for TOPOPT and VERIFY (56.9 and 69.4 percent fewer coherency cycles, respectively). In both cases realistic simulations indicated the reverse. Berkeley Ownership provided better coherency performance than the Firefly for CELL and SPICE (49.4 and 52.3 percent fewer cycles, respectively), but did less well for TOPOPT and VERIFY (114.5 and 54.1 percent more coherency cycles than Firefly).

This discrepancy between the realistic simulations's bus operations and the model's state transitions occurs whenever the coherency block in the sharing analysis does not match that in

the real machine. In Berkeley Ownership, the unit of invalidation and reread is an entire cache block[10]; and the block size in SPUR is 32 bytes. The effects of SPUR's large coherency block overshadow the coherency overhead due to the intrinsic sharing pattern in the applications. The Firefly results more closely correspond to those of the write run model prediction, primarily because the coherency block is identical in both the model and the realistic simulations.

The effects of the coherency block produce either a savings or an additional coherency cost, depending on the inter-processor memory access pattern to words within the blocks. This pattern can be characterized by two distinct modes of behavior. In *sequential sharing*, a processor makes multiple writes to the words within a block, uninterrupted by accesses from other processors. In *fine-grain sharing*, multiple processors contend for one or more words within the block, and the number of per-processor sequential writes is very low.

Whether a program exhibits sequential or fine-grain sharing affects the amount of coherency overhead incurred. In write-invalidate protocols sequential sharing reduces coherency overhead by decreasing both the number of invalidations and the number of invalidation misses. Conversely, when there is fine-grain sharing, the number of invalidations and invalidation misses is higher. For both types of memory reference behavior, the larger the coherency block, the more pronounced the effect on coherency overhead (see Chapter 5).

Sequential sharing can benefit both the writer and the readers of a cache block containing a shared address (See Figure 4-6). For example, after an invalidation, a writing processor possesses the only cached copy of the block. It pays the coherency overhead (the invalidation signal) for the first write to the block, but can update the remaining words without additional bus operations. In contrast, the write run model records a separate write run for each word within the block. Therefore the invalidating signal is counted for the initial write to *each* word

[10] Although coherency block and the cache block are synonymous for write-invalidate in these simulations, I shall continue to use the term "coherency block" to underscore its semantics.

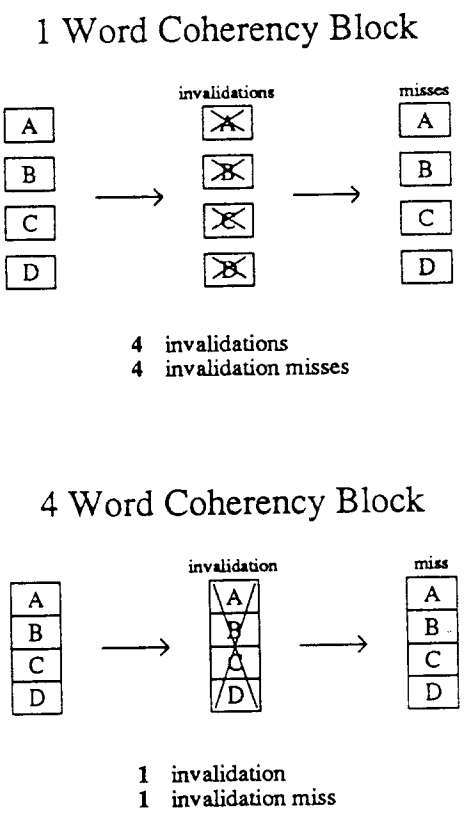

Figure 4-6: Sequential Sharing

This figure illustrates how sequential sharing for shared data within a large coherency block can reduce coherency-related bus operations. In both the one-word and four-word diagrams each address is written by one processor and read by another. The penalty for the one-word coherency block is worse than that for the four-word, by a factor equal to their size difference. (The arrows move in the direction of time).

in the block, rather than just once, and the spatial locality of reference for shared data within the coherency block is missed. An analogous situation exists for the readers. In this case the invalidation miss penalty is paid only for the first read to the block. All other reads are cache hits, and are free of coherency overhead. In CELL and SPICE, sequential sharing for the write-shared data within the 32 byte coherency block decreased both the number of invalidations and rereads.

On the other hand, contention for a particular address within a block (fine-grain sharing) produces more invalidations that interrupt all processors' use of the data in the block and a corresponding increase in the number of invalidation misses to get it back. The greater the number of processors contending for an address, the greater the number of invalidation misses.

The problem is exacerbated with a large block size, because contention can occur for *any* of the addresses in the block. Alternating writes by different processors to the different words within a block produce separate invalidations for each write (see Figure 4-7). The invalidations are responsible for a subsequent rise in invalidation misses. The invalidation misses occur each time a processor rereads any word in the block. The overhead is paid even when the processor reads an address that was not updated.

Reads by *different* processors to the words within an invalidated block also contribute to the rise in invalidation misses (see Figure 4-8). Recall that an invalidation to one word in a block causes all other words to be nullified; when the subsequent reads to these addresses are issued by different processors, additional read misses are incurred to get them back.

In the write run model there are separate write runs for each word in the block, and the writes to one address do not affect the reads to another. In fine-grain sharing that affects both readers and writers, accesses that are read misses in the realistic simulations are considered hits in the modeling analysis, and consequently are not counted as coherency overhead.

Fine-grain sharing was prevalent in the remaining two traces, TOPOPT and VERIFY. Recall that the average write run length for shared addresses in TOPOPT and VERIFY was 5.13 and 6.37, respectively, higher than the other two traces by a substantial margin. With the one-word coherency block of the sharing analysis, only the first of the writes in these runs caused an invalidation. However, in the realistic simulations most of the writes caused invalidations, because of the interleaved (by processor) accesses within the larger (32 byte) coherency blocks.

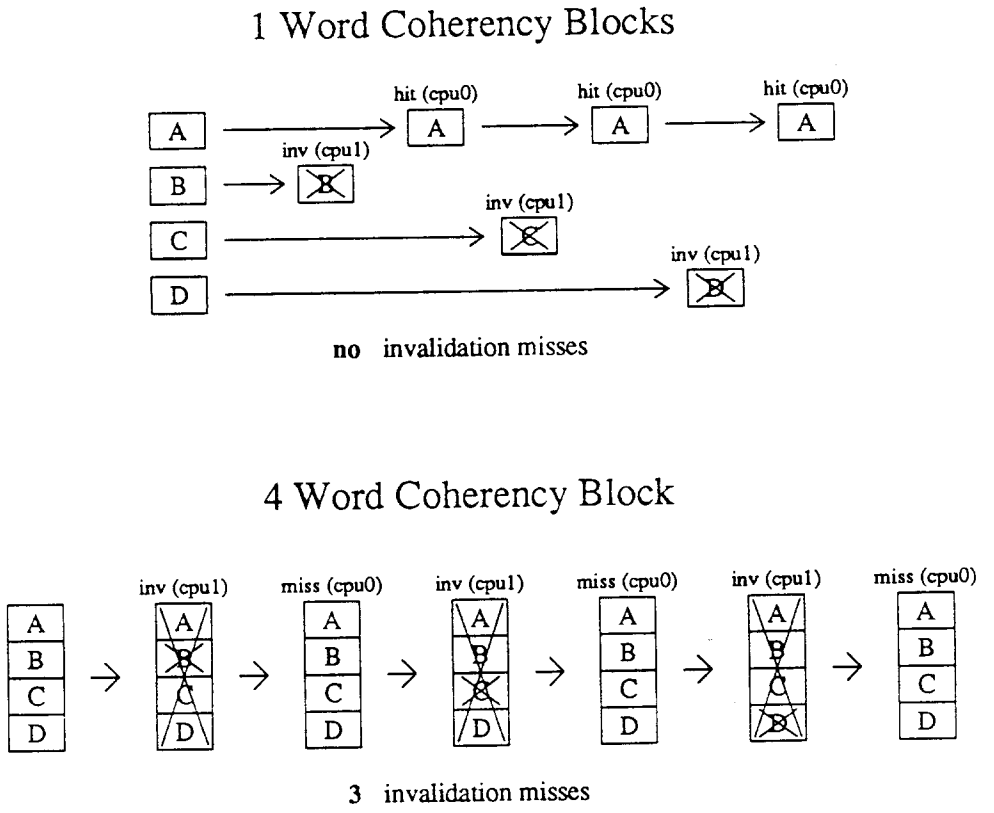

Figure 4-7: Fine Grain Sharing (for Writers)

This illustration of fine-grain sharing depicts the effects of inter-processor *write* activity for some addresses in a coherency block on others. In the one-word coherency block (used in the write run model) the writes to addresses B through D do not affect reads to A; in the four-word coherency block they cause invalidation misses for each reread, because they invalidate the entire block. (The arrows move in the direction of time).

4.8. The Coherency Block Write Run Model

The comparisons of coherency overhead between actual simulations and the architecture- and protocol-independent write run model have demonstrated that the model is too general for write-invalidate protocols. In order to obtain model predictions that more accurately reflect actual coherency costs, I incorporated the size of the coherency block into the otherwise architecture-independent write run model. In the new *coherency block model*, the unit of (write

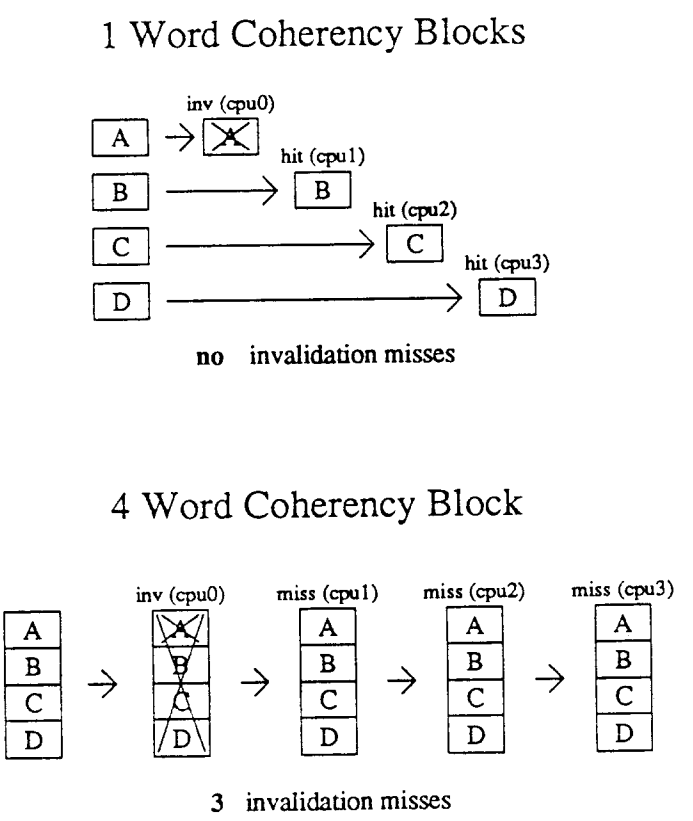

Figure 4-8: Fine Grain Sharing (for Readers)

This diagram of fine-grain sharing demonstrates inter-process *read* contention for different addresses within a coherency block. For large coherency block sizes, the contention causes additional invalidation misses. With the one-word coherency block of the write run model, the invalidation misses become cache hits. (The arrows move in the direction of time).

run) analysis is the entire coherency block, rather than the one-word CPU access. All other assumptions of the architecture-independent model still remain in effect.

The coherency block write run model produced much more accurate approximations. First, for all traces the model correctly predicts the protocol with less coherency overhead (Berkeley Ownership for CELL and SPICE, and the Firefly for TOPOPT and VERIFY) (see Table 4-12). Furthermore, the absolute magnitude of the predictions is quite close to actual per-

Cost of Berkeley Ownership & Firefly in the Coherency Block Write Run Sharing Model						
Trace	Coherency Protocol	Different Write Run (arcs)	Same Write Run (arcs)	End of Write Run (arcs)	Coherency Overhead (cycles)	Normalized to Berkeley Ownership
CELL	**Berk. Own.**	10091	39305	12447	345138	1.00
	Firefly				543356	1.57
SPICE	**Berk. Own.**	4493	24358	3760	121596	1.00
	Firefly				317361	2.61
TOPOPT	Berk. Own.	7957	1607	5164	188436	1.00
	Firefly				105204	.59
VERIFY	Berk. Own.	16912	20258	26094	672636	1.00
	Firefly				408870	.61

Table 4-12: Coherency Block Write Run Model Comparison of Berkeley Ownership & Firefly

This table depicts the number of occurrences of each arc in the coherency block write run model. The total number of cycles is obtained by multiplying the frequency of each arc transition and the arc costs in Table 4-8. The bold entries indicate which of the protocols had less coherency overhead; these predictions match the outcome of realistic simulations. Notice that for the Firefly the total number of cycles is almost identical to the architecture-independent results. For both models the total number of bus operations is the same, although apportioned differently between the Different Write Run and Same Write Run states. Under write-broadcast, the transitions to both states produce the same bus operations. Therefore the cost of all operations and consequently total coherency overhead is identical in both models.

Comparison of Realistic Simulation to Models					
Trace	Coherency Protocol	Architecture Independent Model (percent)	Coherency Block Model (percent)	Algorithmic Use Variation (percent)	Finite Cache Variation (percent)
CELL	**Berk. Own.**	-72.29	8.32	4.80	4.47
	Firefly	0.00			
SPICE	**Berk. Own.**	-154.20	9.69	7.37	2.86
	Firefly	0.30			
TOPOPT	Berk. Own.	79.91	16.51	-1.17	-2.32
	Firefly	0.00			
VERIFY	Berk. Own.	81.11	-1.54	-2.83	-2.83
	Firefly	-0.04			

Table 4-13: Coherency Overhead: Comparison of Realistic Simulations to the Write Run Models

This table compares coherency overhead predictions of progressively more architecturally detailed write run models to the actual overhead in the results from realistic simulations. The results are discussed in detail in Section 4-9. The bold font indicates which protocol had the lower coherency overhead in the realistic simulations.

formance (see the second column of figures in Table 4-13). The percentage difference between the coherency block model's results and those of actual simulations is 8.32 and 9.69 percent for the traces with sequential sharing (CELL and SPICE, respectively) and 16.51 and -1.54 percent for those that exhibited fine-grain sharing (TOPTOP and VERIFY).

The revised write run metrics also support the coherency block model's relative protocol performance predictions. As mentioned above, CELL and SPICE have fewer coherency-related cycles with Berkeley Ownership than Firefly. This is reinforced by their longer average write run lengths (4.90 for CELL and 6.42 for SPICE, as opposed to 1.20 for TOPOPT and 2.20 for VERIFY) and a broader write run length distribution (see Table 4-14). The majority of write runs still terminate with zero or one external rereads (see Table 4-15). The exact figures are 76.1 percent for CELL, 92.6 percent for SPICE, 89.7 percent for TOPOPT and 67.3 percent for VERIFY. Like the original architecture-independent write run model, these figures are low enough that the write run length, rather than the number of external rereads, determines the model's relative protocol predictions.

The new sharing ratios for TOPOPT and VERIFY are extremely high (28.5 and 21.5, respectively), indicating considerable contention for the write shared data (see Table 4-16). Since write-broadcast protocols perform well during periods of contention, it is not surprising that both the coherency block model and the realistic simulations indicate that the Firefly is the better protocol for these traces. For three of the traces, the number of busywaiters is identical to the architecture-independent results. The lone exception is CELL, whose figures differ, but only slightly. (The percentage of locks that were unlocked with no other processor waiting dropped from 86.6 to 85.8 percent.) The similarity in lock activity between the two models demonstrates that locks have not been allocated to the same coherency block. The very sequential sharing for locks implies that the contention exhibited by the high sharing ratio for TOPOPT and VERIFY pertains only to the applications shared data. (In fact, recall that

Coherency Block Write Run Length Histogram								
Run Length Bins	Traces							
	CELL		SPICE		TOPOPT		VERIFY	
	% Write Runs	% Writes	% Write Runs	% Writes	% Write Runs	% Writes	% Write Runs	% Writes
1	28.0	5.1	11.8	1.5	91.5	74.0	79.3	34.7
2	25.8	9.3	31.5	8.0	6.8	11.1	14.4	12.6
3	10.3	5.6	0.8	0.3	1.1	2.6	2.4	3.2
4	19.3	13.9	14.5	7.3	0.2	0.7	1.0	1.8
5	2.2	2.0	1.2	0.8	0.0	0.2	0.7	1.6
6	3.0	3.3	7.1	5.4	0.0	0.1	0.6	1.7
7	1.2	1.6	0.0	0.0	0.0	0.2	0.2	0.5
8	2.9	4.2	17.8	18.1	0.0	0.0	0.1	0.5
9	0.7	1.1	0.0	0.0	0.0	0.2	0.1	0.2
10	0.7	1.3	0.1	0.2	0.0	0.0	0.1	0.5
11	0.2	0.4	0.0	0.0	0.0	0.1	0.0	0.0
12	0.9	2.0	2.2	3.4	0.0	0.0	0.2	1.2
13	0.1	0.3	0.0	0.0	0.0	0.1	0.1	0.3
14	0.3	0.9	0.8	1.5	0.0	0.0	0.0	0.2
15	0.1	0.3	0.0	0.0	0.0	0.3	0.0	0.2
16	0.3	0.8	4.3	8.7	0.0	0.0	0.0	0.2
17	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.1
18	0.2	0.7	0.6	1.3	0.0	0.0	0.0	0.1
19	0.1	0.2	0.0	0.0	0.0	0.2	0.0	0.1
20	0.4	1.4	0.3	0.8	0.0	0.0	0.0	0.3
>20	3.1	33.8	6.9	24.2	0.2	7.5	0.4	35.9
Total Write Runs	10091		4493		7957		16912	
Avg. Write Run Length	4.90		6.42		1.20		2.20	
Total Writes		55969		35403		9837		38724

Table 4-14: Length of the Coherency Block Write Runs

This histogram depicts the percentage of write runs that have a particular write run length for the coherency block model, assuming a 32 byte coherency block. Unlike the architecture-independent model, the programs with sequential sharing (CELL and SPICE) have longer average write run lengths, and their distribution of write run lengths is spread more evenly. The histogram also contains the percentage of total writes that are in a write run of a given length. The percentages support the average write run length data.

Coherency Block External Rereads Histogram				
External Rereads Bins	Traces			
	CELL %	SPICE %	TOPOPT %	VERIFY %
0	19.3	8.0	81.2	7.8
1	56.8	84.6	8.5	59.5
2	12.8	2.6	2.7	16.2
3	4.3	2.3	1.2	8.0
4	2.0	2.5	0.6	4.2
5	1.3	0.0	1.1	2.3
6	1.3	0.0	0.6	1.0
7	0.8	0.0	1.2	0.5
8	0.8	0.0	0.1	0.3
9	0.4	0.0	0.1	0.2
10	0.2	0.0	2.7	0.0
11	0.0	0.0	0.0	0.0
12	0.0	0.0	0.0	0.0
Total External Rereads	9194	3524	7841	16515
Avg. External Rereads	1.35	1.07	0.66	1.58

Table 4-15: Number of External Rereads Following a Write Run
(Coherency Block Model)

This histogram depicts the percentage of write runs that were followed by a particular number of external rereads. Like the architecture-independent write run model, the average number of external reads is low and a single or no external rereads terminates most of the write runs in all traces. Again, this means that there was little or no contention for the shared variables, and, *in isolation*, argues for a write-invalidate coherency protocol.

Sharing Ratio		
Trace	Value	Time Period (mem refs in 000's)
CELL	7.4	3,732
SPICE	4.6	1,538
TOPOPT	28.5	3,300
VERIFY	21.5	3,604

Table 4-16: Sharing Ratio (Coherency Block Model)

Contention for the write shared addresses in the TOPOPT and VERIFY is fairly high. This is in sharp contrast to the almost lack of contention predicted by the architecture-independent model (1.8 and 1.9, respectively). Again, the time period is measured by the total number of references processed by each processor.

TOPOPT uses no locks.)

4.9. Refining the Model

Although the coherency block write run model greatly improved the accuracy of the coherency overhead cycle predictions, they are not yet acceptably close to actual values. Additional improvements can be realized by further refining the model. The changes involve a more detailed cost analysis for the transition to the Different Write Run state and dropping the infinite cache assumption.

Recall that total coherency overhead for both write run models was calculated by multiplying the number of state transitions by protocol-specific arc costs. The state transitions signify only that a change of write run state took place, without stipulating the type of bus operation that implemented them. The arc costs assume the bus operation that occurs in the most common case. For write-invalidate protocols, this is an invalidation signal to the Different Write Run state, a data transfer to the End of Write Run state and no bus operation to the Same Write Run state. In realistic simulations, these bus operations do not always take place, both because of algorithmic program behavior and the constraints of a finite cache.

The stipulation that an invalidation signal be the cost of a transition to the Different Write Run state is based on the assumption that write references result in cache hits. When the writes produce cache misses, the actual cost is that of the more expensive data transfer. Under write-invalidate protocols, write misses to shared data have several causes. First, the algorithm of the program may dictate that the program's first access to the data is a write, rather than a read.[11] Second, invalidations issued by other processors will nullify cache blocks. And finally, blocks

---

[11] Even if the program's first access to the data is a read, the operating system's handling of page faults could cause the first *cache access* to a shared address to be a write. For example, when the virtual space for the program is first allocated, the kernel could zero-fill the heap pages in user space. These writes would generate data transfers rather than invalidation signals. If an address was still in the cache when the program referenced it, its cache block would already be private, and the program's reads and writes would cause no additional bus operations.

Other scenarios can be constructed in which different coherency operations are generated. In Sprite [Oust88], the (virtual) pages are zero-filled in kernel space, flushed from the cache, and then mapped to user space. In this case a program's first read to shared data will miss in the cache, and the subsequent write will produce an invalidation signal.

may be evicted through block replacements. Although the caches used in the realistic simulations are moderately large relative to the size of the working set of the programs, block replacements do have a small effect on coherency overhead. Block replacements also cause additional cycles to be charged for transitions to the Same Write Run state. Here either an invalidation signal or a data transfer replaces the assumed no bus operation.

Table 4-13 illustrates the progression of coherency-related bus cycles, as the coherency block model is altered to account for the above factors. In all cases the models' coherency overhead cycles were compared to those in the realistic simulations. The first two columns indicate the percentage differences for the architecture-independent and coherency block models. Both model figures reflect coherency cycles, based solely on the number of transitions and the general arc cost assumptions. The column "Algorithmic Use Variation" uses a weighted average between invalidation signals and data transfers for the cost to the Different Write Run state. For the latter operation write runs began with write misses, caused either by a program's first access to shared data being a write or invalidations by other processors. The fourth column, "Finite Cache Variation", includes the effects of block replacements in the 128K byte cache. For this model, additional cycles are charged for data transfers on transitions to the Different Write Run state and both data transfers and invalidation signals for transitions to the Same Write Run state.

Because the coherency block model abstracts out the differences in the costs of several transitions, it tends to give optimistic predictions of actual protocol behavior. For the same write run transition frequencies, total coherency overhead, measured by the type of bus operation that actually occurred (in the realistic simulations), is higher. Therefore, as the variations are progressively included in the model, the percentage difference between the actual coherency cost and the model predictions should become increasingly less. This is the case for the traces with sequential sharing, CELL and SPICE. In reality, coherency overhead figures are perturbed by additional factors not represented by the detailed cost analysis of the state transitions. One

example is the bus arbitration protocol. All write run model variations assume a round robin protocol that ensures equal access to the bus for all processors. On the other hand, the NuBus protocol used in the realistic simulations guarantees fair access within a wave of requests, but gives higher wave entrance priority to certain processors at high bus utilization levels. In particular, the CPUs with the lowest two identification numbers will each be shut out of approximately half of the waves [Vern88]. Biasing the order of processor bus procurement in this way changes the global sequence of shared references. The consequence of the unfairness could be that processors with greater access to the bus will be able to process all references, including those to shared data, at a faster rate than those processors with a lower priority to the bus. They will therefore be able to complete more accesses to shared data between intervening references by other processors. Consequently, they will tend to have fewer write runs with longer average write run lengths. Both factors will tend to decrease the total number of coherency overhead cycles for write-invalidate protocols, perhaps even below the models' predictions.

This very likely explains the results for VERIFY, where the coherency block model predicted more overhead than was reported in the realistic simulations. Once the lower model base was established, the model improvements from the detailed state transition analysis and the finite cache widened the gap. In VERIFY's realistic simulation, bus utilization was 96.6 percent; therefore the biased bus arbitration described above would apply. Write run metrics for the traces also support this analysis; on average there were fewer write runs in the realistic simulations than for the coherency block model, and the average write run length was longer.

TOPOPT also had fewer and longer write runs in the actual simulations. However the actual coherency overhead cycles were not close enough to those in the models to cause the latter to dominate, until the model that incorporated a detailed breakdown of the transitions that began write runs (the algorithmic use variation). Here the large drop in the comparison figures between the coherency block and algorithmic use variation (16.51 to -1.17 percent) is attributable to the large proportion of write runs that began with a write (84 percent). In the

algorithmic use variation these transitions incurred the cost of a full data transfer.

4.10. Chapter Summary

The results in this chapter have demonstrated the limitations of simple, architecture-independent sharing models in accurately predicting coherency overhead for write-invalidate protocols. Actual performance for these protocols depends on several architecture- and program-dependent factors, the most important of which is the coherency block. When the coherency block is larger than the one-word unit of access in the architecture-independent model, the memory access pattern to the shared data within the block dominates the effects of the sharing pattern intrinsic to the algorithm of the program. A savings in coherency overhead occurs when the memory access pattern is one of sequential sharing; and additional coherency cycles result with fine-grain sharing.

When the size of the coherency block is incorporated into the write run model, model predictions reduce the relative error (between the model and realistic simulations) by a factor of *4.8 to 52.7*, depending on the trace. Model predictions come within an average of 9 percent of actual simulations. The coherency unit model is more accurate, because it includes two factors that are crucial to modeling parallel program activity. First, it reflects locality of reference to shared data in the workload through the write run characterization. Second, it bases the locality analysis on the size of the coherency unit. Incorporating the coherency unit into the model places a limit on the per-processor locality. The limit reflects the inter-processor activity for shared data that occurs in a running multiprocessor.

Integrating all factors (the coherency block, a detailed analysis of the state transitions that begin a write run and a finite cache) produced model predictions that were close to realistic simulations. The final discrepancies ranged from 2.3 to 4.5 percent. The cost of incorporating the factors is the necessity to redo the modeling simulations should any of the factor values change.

The original, architecture-independent model is still useful for several reasons. First, it provides accurate predictions for write-broadcast protocols. This is primarily because the size of their coherency block matches the unit of analysis in the model and all coherency-related bus operations have the same cost. In addition, cache misses have no effect on coherency overhead for write-broadcast, because updates to shared data are only broadcast on cache hits. When a block is replaced or a write run begins with a write, the block is read into the cache before the broadcast is issued. The same amount of coherency overhead is incurred, if the block had not been replaced or if the write run had begun with a read. The infinite cache in the architecture-independent model perturbs coherency overhead for write-broadcast slightly, in that it guarantees that once data is shared, it remains shared. In the realistic simulations shared data is occasionally replaced, and not referenced. The shared bus line is eventually dropped, and write broadcasts are discontinued. Infinite cache effects would have been much more pronounced had the cache in the realistic simulations been smaller.

Secondly, by concentrating totally on write run activities, the architecture-independent write run model highlights the differences between the two types of protocols and explains how different patterns of sharing affect relative protocol performance. Third, the architecture-independent model was used to isolate the factors that turned out to be important in modeling write-invalidate coherency costs, previously mentioned. And lastly, it requires only one simulation per trace, for all cache architecture parameters.

4.11. References

[Agar88] A. Agarwal and A. Gupta, "Memory-Reference Characteristics of Multiprocessor Applications under MACH", *Proceedings of the 1988 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 16, 1 (1988), 215-225.

[Dare87] F. Darema–Rogers, G. F. Pfister and K. So, "Memory Access Patterns of Parallel Scientific Programs", *Proceedings of ACM SIGMETRICS*, 15, 1 (May 1987), 46-58.

[Hill86] M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. Ousterhout and D. A. Patterson, "SPUR: A VLSI Multiprocessor Workstation", *IEEE Computer*, 19, 11 (November 1986), 8-22.

[Katz85] R. Katz, S. Eggers, D. Wood, C. L. Perkins and R. Sheldon, "Implementing a Cache Consistency Protocol", *Proceedings of the 12th Annual International Symposium on Computer Architecture*, 13, 3 (June 1985), 276-283.

[Oust88] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson and B. B. Welch, "The Sprite Network Operating System", *IEEE Computer*, 21, 2 (February 1988), 23-36.

[Smit85] A. J. Smith, "Cache Evaluation and the Impact of Workload Choice", *Proceedings of the 12th Annual International Symposium on Computer Architecture*, 13, 3 (June 1985), 64-73.

[Thac88] C. P. Thacker, L. C. Stewart and E. H. Satterthwaite, Jr., "Firefly: A Multiprocessor Workstation", *IEEE Transactions on Computers*, 37, 8 (August 1988), 909-920.

[Vern88] M. K. Vernon and S. T. Leutenegger, "Fairness Analysis of Multiprocessor Bus Arbitration Protocols", Technical Report #744, Computer Sciences Department, Univ. of Wisconsin-Madison (September 1988).

5 The Effect of Sharing on the Cache and Bus Performance of Parallel Programs

5.1. Introduction

The cache behavior of uniprocessor programs, in particular, the effect on performance of changing cache parameters, has been extensively analyzed (e.g., [Agar88, Alex86, Good87, Hill87, Przy88, Smit85, Smit87]). For small and medium sized caches, increasing the cache size causes a drop in the miss ratio that is substantial enough to reduce the effective memory access time, despite the additional cache access time of the larger caches. For very large caches, the miss ratio still falls; however, its value is quite low. The resulting high hit ratio, multiplied by the longer cache access time, causes the effective access time to rise. The miss ratio trend for increasing block size is not as consistent. A larger block size also reduces miss ratios, but only up to a certain size. After reaching this memory pollution point, miss ratios begin to climb. But even the declining miss ratio does not always increase performance, because of the additional bus traffic latency caused by the larger transfer block.

Miss ratios and bus utilization of parallel programs should be higher than those of uniprocessor programs, because additional bus traffic is required to maintain coherent caches. This is

a critical performance issue in single-bus multiprocessor design, since bus bandwidth is the limiting performance factor in such a system. If the cache and bus behavior of parallel programs, varying across cache and block sizes, is radically different from uniprocessor programs, then new rules of thumb are needed to design memory systems for multiprocessors.

The goal of this chapter is twofold: first, to analyze the cache and bus behavior of parallel programs running under write-invalidate coherency protocols; and second, to compare this behavior to that of their uniprocessor counterparts. The research shows that parallel programs do have different cache and bus behavior than uniprocessor programs; and that it is the references to shared data that are responsible for the difference. The results are most dramatic when increasing block size. Here the proportion of sharing-related misses to total misses rises. The consequence is a higher miss ratio than for uniprocessor programs. For some traces the effect was great enough to cause miss ratios to rise with increasing block size, rather than fall. Sharing also worsens the miss ratios when increasing cache size; again, the effect is more pronounced with larger cache sizes. For most programs sharing-related bus traffic dominates bus utilization cycles with large caches (128K bytes and up) and all block sizes studied (4 to 32 bytes). At these cache configurations it is the sharing traffic that creates the multiprocessor bus bottleneck.

These results indicate that larger caches and block sizes, the traditional techniques for improving cache performance, are less effective with parallel programs than uniprocessor programs. However, additional performance improvements can still occur using software techniques. For the programs analyzed, the amount of sharing overhead depended on the intra-block memory reference pattern for shared data. Programs that exhibited sequential sharing performed better than those with fine-grain sharing. If programmers (or compilers) are aware of memory reference patterns when writing (generating) parallel code, they can attain better program performance by altering the memory organization of the shared data.

The remainder of the chapter contains the studies of the cache (section 5.2) and bus (section 5.3) behavior of parallel programs, each investigating the effects of changing both block and cache size. Section 5.4 summarizes the results; the summary discusses the implications of cache and bus performance of the parallel programs, both for multiprocessor cache design and software design.

5.2. The Effect of Sharing on Miss Ratios

5.2.1. Varying Block Size

Cache miss ratio studies of uniprocessor programs have indicated that for a fixed size cache, the miss ratio initially drops as the block size of the cache increases [Agar88, Alex86, Good87, Hill87, Przy88, Smit87]. The decline is due to improved cache hits because of locality of reference. However, as block size continues to increase, the decrease in the miss ratio tapers off. For small and medium sized caches, those in the range of 4K bytes to 16K bytes, the miss ratio decline may terminate at some particular block size (in [Good87], it is 32 bytes and 128 bytes, respectively), after which the miss ratio begins to rise. The termination is known as the memory pollution point.[1] As cache size grows, the pollution point shifts to an increasingly larger block size. For 128K byte caches, [Agar88] reports that the pollution point is not reached with block sizes up to 32 bytes (the configurations in this study). Therefore, for uniprocessor programs, miss ratios should continue to decline until that point.

Analysis of the parallel traces indicates that their miss ratios do not always follow the trend of uniprocessor programs (see Figure 5-1). CELL and SPICE consistently exhibit the expected decline; but the miss ratios for TOPOPT and VERIFY actually *increase*. Their rise is

[1] For a fixed sized cache, a larger block size results in fewer cache lines. The pollution point occurs because memory references take place to noncontiguous data that do not reside in the cache, while contiguous, but unreferenced, data remain in the larger block. Until the pollution point is reached, the larger block size implicitly prefetches data that will be referenced in the near future.

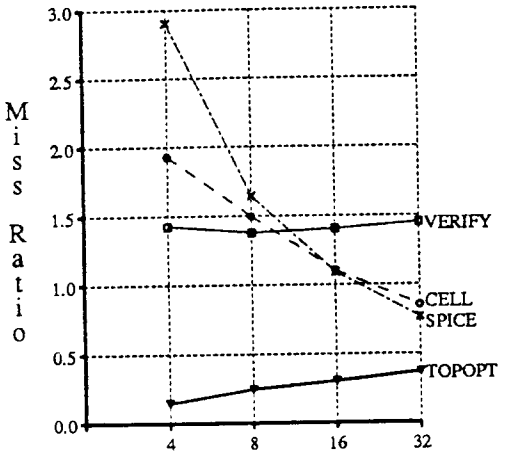

Figure 5-1: Miss Ratio

Parallel traces exhibit two miss ratio trends as block size increases. Miss ratios decline for programs with sequential sharing (CELL and SPICE); however, for programs with fine-grain sharing (TOPOPT and VERIFY), they are dominated by the misses caused by intra-block contention for shared data, which produce the rising curves. (All block size graphs are for a 128K byte cache.)

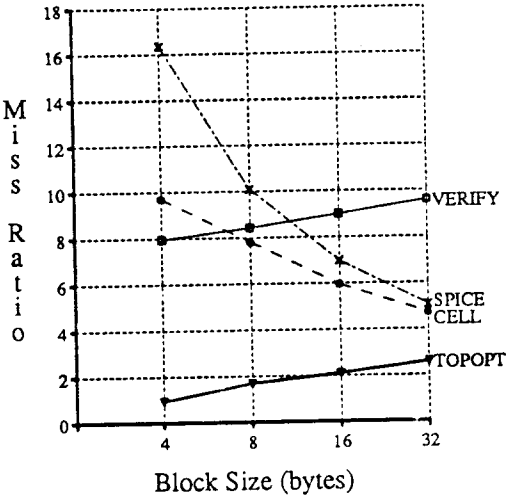

Figure 5-2: Shared Miss Ratio

The shared miss ratios (misses to shared data divided by total references to shared data) follow the trend of the total miss ratios, but have a value that is 5 to 7 times higher. The higher figures reflect the poorer locality of shared data.

slight, and, for one of the traces, not continuous across all block sizes. However, their behavior was completely unexpected, given the uniprocessor literature. Miss ratios for the shared references only,[2] depicted in Figure 5-2, indicate almost identical behavior, but at higher values. Due to the poorer locality of reference for shared data, miss ratios for shared data were 5 to 7 times greater than total miss ratios, depending on the particular trace and block size.

Eliminating invalidation misses from the miss ratio calculations[3] leaves a uniprocessor component that approximates uniprocessor miss ratios. The uniprocessor miss ratio component

[2] The shared miss ratio is the number of misses to shared data divided by total references to shared data.

[3] The new calculation is: $\frac{total\ misses - invalidation\ misses}{total\ refs}$.

for all traces corroborated results from previous cache studies of uniprocessor programs. In other words, the uniprocessor miss ratio component declined as block size increased, and the marginal rates of decline also decreased with block size (see Figure 5-3). The values were less than the multiprogramming miss ratios reported in [Agar88]; however, this is to be expected, since these traces contain applications references only, and the traces in [Agar88] include operating systems references. The uniprocessor miss ratio components for SPICE are most typical of the results of the composite uniprocessor applications workload reported in [Good87].

The predictable trend of the uniprocessor component of the miss ratios suggests that it is the invalidation misses that are responsible for the variable miss ratio behavior of the parallel

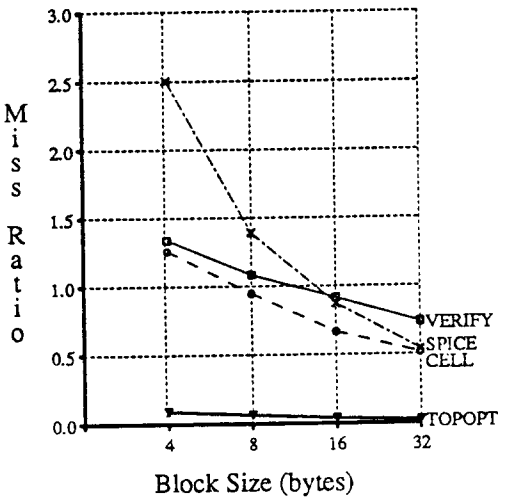

Figure 5-3: Uniprocessor Component of the Miss Ratio

The uniprocessor component of the miss ratio for parallel programs mimics the declining miss ratio behavior of uniprocessor programs. This suggests that the variability of the miss ratio curves for parallel programs is caused by the invalidation misses.

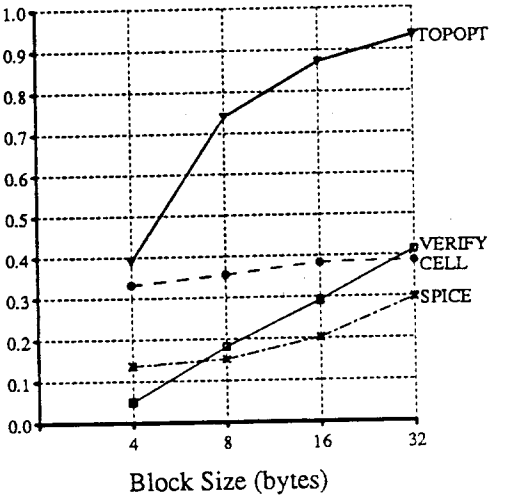

Figure 5-4: Ratio of Invalidation Misses to Total Misses

The ratio of invalidation misses to total misses increases as block sizes increase. At larger block sizes the invalidation misses of three of the traces comprise a substantial portion of the total; and for TOPOPT they dominate miss ratio behavior. (The numbers are the geometric mean of the ratio of invalidation to total misses, across all processors.)

programs. A more detailed examination reveals that two interacting factors determine the miss ratio trends (see Figure 5-4). First, as block size increases, invalidation misses become a larger fraction of total misses. Therefore they become an increasingly significant determinant of miss ratio behavior. The second factor is the high value of this fraction at the larger block sizes. At small block sizes, the uniprocessor misses dominate. However, at larger block sizes the number of invalidation misses is either a substantial (CELL, VERIFY and SPICE) or overwhelming (TOPOPT) proportion of the total. The combination of these factors forces the miss ratios to follow the trend of the invalidation misses as block size increases. For many block sizes the invalidation misses are the single most determining factor in miss ratio behavior.

The traces exhibited two distinct invalidation miss trends.[4] For programs whose memory reference pattern for shared data is dominated by sequential sharing, such as CELL and SPICE, the number of invalidation misses declines as block size is increased (see Figures 5-5 and 5-6). Shared data in these traces have good spatial locality of reference. Each processor tended to read several contiguous words in succession, all of which had been previously invalidated. With the 32 byte block size, the invalidation miss penalty was incurred only for the first of eight words; with smaller blocks, for example, 4 bytes, it was incurred for each. Because the invalidation miss trend reinforced that of the uniprocessor miss ratios, the miss ratios declined. SPICE, in particular, had good locality of reference. Its shared data structures had been sized to the ELXSI 6400 64 byte cache block, explicitly to avoid fine-grain sharing for addresses within the block. Therefore for block sizes considered in this study (up to 32 bytes), little contention was observed.

For programs with fine-grain sharing within a block, such as TOPOPT and VERIFY (see Figures 5-7 and 5-8), the declining uniprocessor miss ratio was offset by the increase in the number of invalidation misses and their large proportion within total misses. Invalidation

[4] The write run results in [Egge88] and Chapter 4, and tracking cache block behavior with the simulator corroborate the difference in behavior between the two groups of traces.

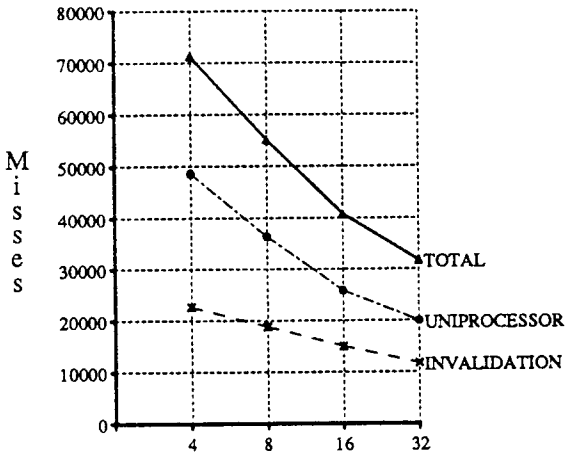

Figure 5-5: Classification of Misses for CELL

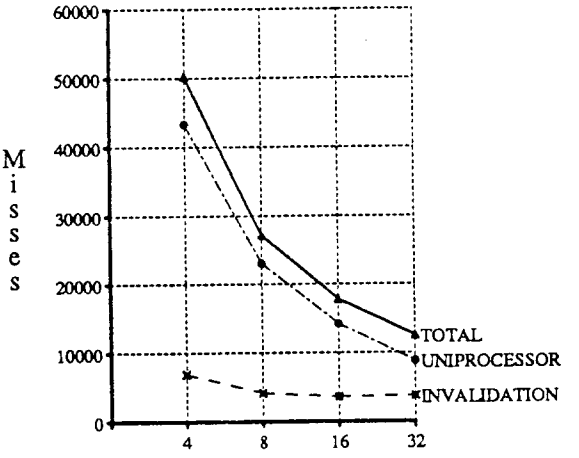

Figure 5-6: Classification of Misses for SPICE

The memory reference pattern of shared data in both CELL and SPICE is one of sequential sharing. Therefore invalidation and uniprocessor misses both decline, producing miss ratio curves that are similar to uniprocessor programs.

misses had the largest effect on TOPOPT, and for two reasons; first, the trace had the most fine-grain sharing, and second, it had a low uniprocessor miss ratio, because its working set fit into the 128K byte cache.

A short note should be made about the miss ratio behavior of the components of shared data, i.e., locks and the shared applications data they protect. For the traces with sequential sharing, the applications data were responsible for the high shared miss ratios depicted in Figure 5-2. Miss ratios for the locks were 2.0 to 4.9 times lower for CELL and 2.4 to 15.8 times lower for SPICE, as block size increased. This corroborates the results in Chapter 4, sections 4.5 and 4.8, that indicate that there was little contention for locks. Greater lock contention would have resulted in more invalidations to them and consequently additional rereads and a higher lock miss ratio. In addition, the lock miss ratio was impervious to increases in block size, indicating that the good locality of reference to shared data in CELL and SPICE was due to the

applications shared data, rather than the locks. The lock miss ratio for VERIFY was more sensative to changes in block size, rising rapidly as block size increased. At 4 bytes it was one-ninth the miss ratio of the applications shared data; at 32 bytes both were comparable.

5.2.2. Varying Cache Size

The benefits of increasing cache size on miss ratios of uniprocessor programs are well known. Numerous trace-driven studies over a variety of workloads have all confirmed that the miss ratio drops as cache size is increased, but that the improvements diminish for large caches [Agar88, Alex86, Good87, Hill87, Przy88, Smit87].

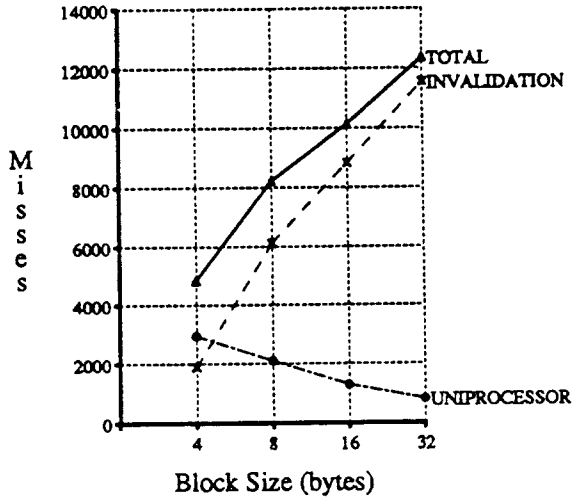

Figure 5-7: Classification of Misses for TOPOPT

TOPOPT is the trace in which the invalidation misses had the most effect and for two reasons: first, it exhibited the most intra-block fine-grain sharing; and, second, its uniprocessor miss ratios were low, because the working set fit into the 128K byte cache.

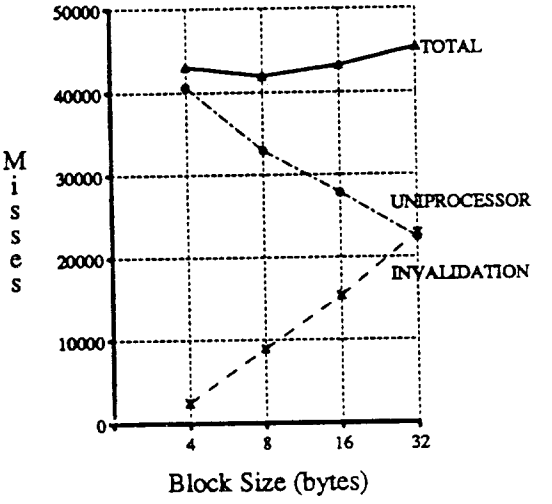

Figure 5-8: Classification of Misses for VERIFY

Although VERIFY's memory reference pattern was one of fine-grain sharing, the uniprocessor misses were proportionately high, particularly at small block sizes. Therefore the miss ratio at first declined, then rose. (Since the individual processor figures for VERIFY were widely skewed, the ratios of invalidation misses to total misses do not match the geometric means in Figure 5-4.)

Shared programs do not experience the same miss ratio benefits of increasing cache size. While it is true that their uniprocessor-related misses decline with larger caches, their invalidation misses either rise or, at best, remain constant. The combination produces a miss ratio that declines with cache size, but is higher than for comparable uniprocessor programs.

The parallel traces support this analysis. For all traces, miss ratios decline with increasing cache size (see Figure 5-9), and total miss ratios are higher than their uniprocessor components (see Table 5-1). The discrepancy increases with cache size, because the uniprocessor miss ratio declines more steeply. The exact figures range from 1.02 to 2.2 higher for SPICE, 1.1 to 2.5 higher for VERIFY, 1.1 to 4.7 higher for CELL and 1.7 to 15.4 higher for TOPOPT, as cache

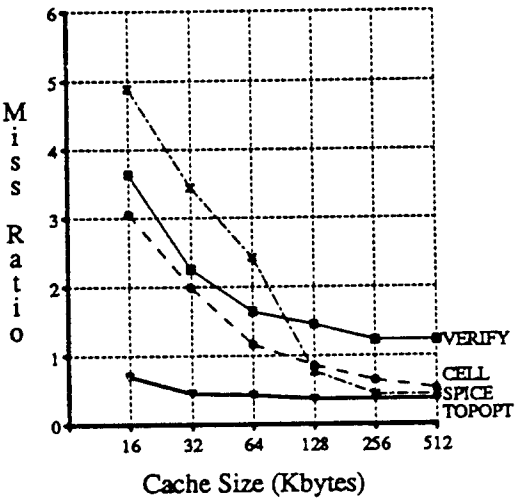

Figure 5-9: Miss Ratio

Increasing the cache size causes the miss ratio for parallel programs to decline. However, the miss ratio is higher than for uniprocessor programs, because of invalidation misses. (All cache size graphs assume a 32 byte block.)

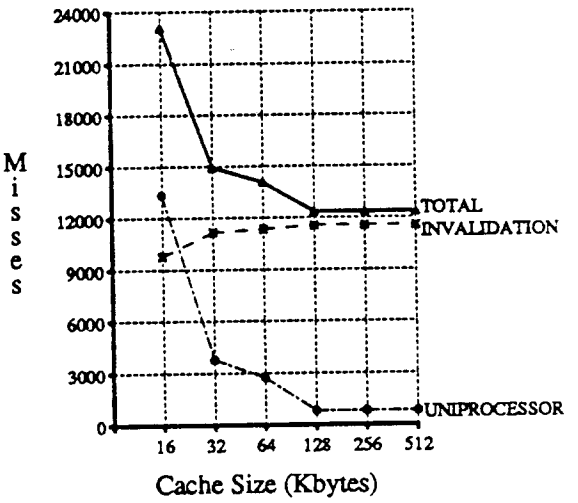

Figure 5-10: Classification of Misses for TOPOPT

Of all the traces TOPOPT has the most fine-grain sharing within the cache block. The effects of that intra-block contention are manifested by the distortion to the miss ratio caused by the number of invalidation misses. (Note that the scale of the y-axis is roughly one-sixth that of VERIFY in Figure 5-11.)

size increases from 16K bytes to 512K bytes. (See comparative curves for different types of misses for two of the traces, TOPOPT and VERIFY, in Figures 5-10 and 5-11, respectively.) These results indicate that the benefits of increasing cache size are less pronounced for parallel programs than uniprocessor programs.

The reason is the presence of invalidation misses. The number of invalidation misses is inversely related to the number of block replacements, i.e., they increase as block replacements fall. At small cache sizes, the number of block replacements is relatively high. If it is assumed

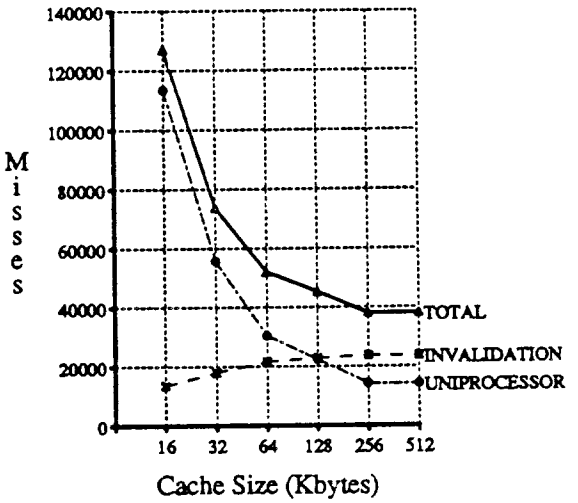

Figure 5-11: Classification of Misses for VERIFY

Miss trends for VERIFY typify the effect of sharing with increasing cache size. The presence of invalidation misses causes the total number of misses (and hence the miss ratio) to be higher than for a uniprocessor program. Their rise, as cache size increases, widens the gap between the total and uniprocessor miss ratio. CELL and SPICE have similar curves, although fewer misses in absolute and fewer invalidation misses proportionately. Their lower figures are due to sequential sharing.

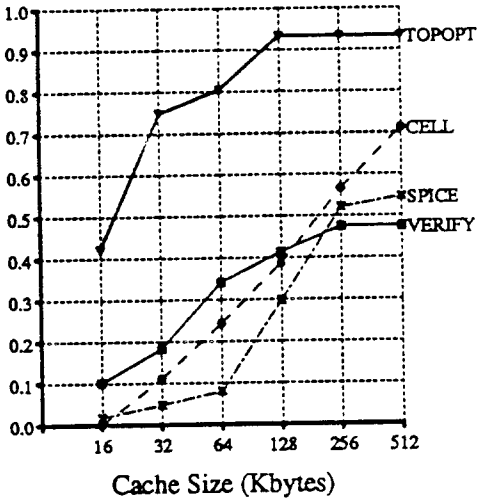

Figure 5-12: Ratio of Invalidation Misses to Total Misses

The ratio of invalidation misses to total misses increases with increasing cache size. The rise is much steeper than with increasing block size for the traces with sequential sharing, CELL and SPICE. (The numbers are the geometric mean of invalidation to total misses, across all processors. Since the individual processor figures for VERIFY were widely skewed, the geometric means do not match the ratios of the absolute misses in Figure 5-11.)

that shared data are replaced at the same rate as private data or instructions, then a proportion of shared data blocks, equivalent to the percentage of blocks replaced, will be eliminated from the cache. They therefore cannot be invalidated and, consequently, will not incur invalidation misses.[5] As cache size increases, the percentage of block replacements drops. Shared data tend to remain in the caches for longer periods of time, have more opportunity to be invalidated, and, consequently, rereferenced via invalidation misses. The number of invalidation misses should be higher with each successively larger cache, approximately by the percentage decrease in block replacements.[6] For very large cache sizes, in which the program's working set fits into the cache, the incremental number of block replacements is negligible, and the invalidations will tend to level off. Again, the traces confirm the analysis. For all traces, the number of invalidation misses rises with increasing cache size. The increase is most pronounced at smaller cache sizes, at which the change in block replacements is also greater (table not shown).

As was true with the block size figures, the proportion of invalidation misses becomes larger as cache size increases (see Figure 5-12). For the traces with sequential sharing and good spatial locality (CELL and SPICE), the effect of the invalidation misses is more pronounced with larger cache sizes than with larger block sizes. Invalidation misses cause the greatest perturbation for TOPOPT, the trace with the most fine-grain sharing. Here the proportion of invalidation misses to total misses ranges from 42 to 93 percent, as cache size increases from 16K to 512K bytes. This causes the total miss ratio to be 1.7 to *15.4* times greater than its uniprocessor component (again, see Figure 5-10).

The working sets of TOPOPT and VERIFY fit into the larger sized caches. Once the caches were filled, the number of uniprocessor misses remained constant. The invalidation

[5] They will, however, like all data and instructions, incur replacement or capacity misses [Hill87]. However, this is a consequence of the smaller cache size, rather than the type of data (shared), and will occur for *all* data and instructions. As caches get larger, some of the capacity misses become invalidation misses. No matter which category they fall in, i.e., no matter what the cache size, they still contribute to the miss ratio.

[6] The rising cost of sharing with larger caches is a problem usually associated with write-broadcast coherency protocols (see Chapter 6); it is interesting that the problem occurs with write-invalidate as well.

Percentage Change in Miss Ratio						
Trace	Miss Ratio Type	Cache Size Spread (in bytes)				
		16K-32K	32K-64K	64K-128K	128K-256K	256K-512K
CELL	Total	-35.193	-41.487	-26.267	-25.147	-18.225
	Uniproc.	**-39.832**	**-50.511**	**-41.174**	**-49.245**	**-54.467**
SPICE	Total	-29.444	-29.770	-68.364	-42.592	-1.324
	Uniproc.	**-31.566**	**-32.189**	**-75.871**	**-61.098**	**-7.120**
TOPOPT	Total	-35.647	-5.812	-12.197	0.000	0.000
	Uniproc.	**-72.467**	**-26.663**	**-70.330**	**0.000**	**0.000**
VERIFY	Total	-38.203	-27.167	-11.596	-15.354	0.000
	Uniproc.	**-47.062**	**-43.031**	**-25.338**	**-35.544**	**0.000**

Table 5-1: Percentage Change in Miss Ratio with Increasing Cache Size

This table contains the incremental miss ratio decline as cache size increases. Note that for all programs and all cache sizes, the uniprocessor miss ratios declined more steeply (bold) than total miss ratios. This indicates that uniprocessor programs obtain a greater benefit from increasing cache size than do parallel programs.

misses also remained constant, because there was no more block replacement effect.

5.3. The Effect of Sharing on Bus Utilization

The critical system bottleneck in a single-bus, shared memory multiprocessor is the bandwidth of the system bus. Relatively few processors can be attached to the bus, unless caching is used to reduce their bandwidth requirements. For a single-bus multiprocessor, the most important consideration for cache organization is how well it limits bus utilization. As was implied by the higher miss ratios in the last section, the bandwidth requirements are greater in parallel programs than uniprocessor programs because of the sharing traffic. With large caches and large block sizes, sharing traffic is expected to dominate the bandwidth and, consequently, dictate the number of processors that can be effectively attached to the bus.

5.3.1. Varying Block Size

Several uniprocessor studies [Good87, Przy88, Smit87] have shown that, up to a certain size, increasing the block size can improve bus performance. A decreasing miss ratio, as block size is increased, is responsible for the improvement. An increase in the time per miss, that also accompanies larger block sizes, partially erodes the benefit of the dropping miss ratio. The breakeven point occurs when the decline in the miss ratio is offset by the increase in the average number of cycles per transfer. Results in any bus utilization study are highly dependent on the cycle assumptions for both memory accessing and bus transfer overhead. But, for caches of the size under study, i.e., 128K bytes, and up to 32 byte blocks, at least one study has shown that the average memory access time declines with increasing block size [Agar88].

Sharing alters bus utilization in two ways. First, invalidation signals and invalidation misses are sources of additional bus traffic, since they do not exist in uniprocessor systems. They cause bus utilization to be higher in parallel programs. Second, the slope of the bus utilization curve is determined by the memory reference pattern to shared data. Programs with fine-grain sharing have miss ratios that increase rather than decrease with block size. Therefore their miss ratios compound the increase in bus traffic caused by the larger transfer unit, and bus utilization increases. For programs that exhibit sequential sharing, miss ratios decline, and the marginal miss ratios (as block size increases) are comparable to those for uniprocessor programs. In this case, bus utilization could proceed in either direction, depending on whether the change in the miss ratio is great enough to offset the increase in the average number of cycles per transfer.

The traces under study reflected these effects. For all traces, bus utilization was higher than its uniprocessor component.[7] (The ranges for the individual traces are: 1.9 to 2.2 higher for

[7] Uniprocessor bus utilization is determined by excluding the cycles used for invalidation signals and invalidation misses.

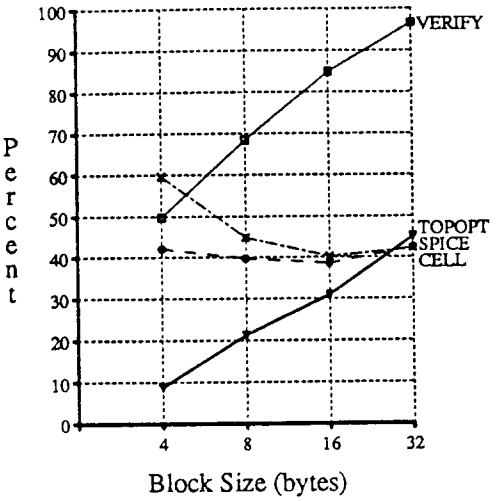

Figure 5-13: Effect of Block Size on Bus Utilization

Bus utilization is calculated as the number of cycles during which a bus operation took place, divided by the total cycles in the simulation. The bus cycles include cycles for the overhead of bus operations, in addition to those counted in bus traffic figures. The sequential sharing of CELL and SPICE produced the declining or flattened bus utilization curves; the fine-grain sharing of TOPOPT and VERIFY exacerbated their already rising average cycles per transfer, resulting in increasing bus utilization. (All block size graphs are for a 128K byte cache.)

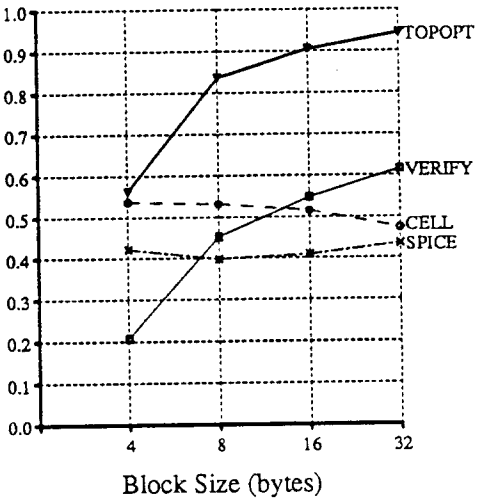

Figure 5-14: Ratio of Sharing Bus Cycles to Total Bus Cycles

The cycles needed for invalidations and invalidation misses were a substantial portion of or completely dominated total bus cycles, over most block sizes. This indicates that efforts to reduce bus bandwidth demands should concentrate on the sharing-related traffic.

CELL, 1.7 to 1.8 higher for SPICE, 2.3 to 17.7 higher for TOPOPT and 1.3 to 2.6 higher for VERIFY.) For the most part, the sequential sharing of CELL and SPICE produced a miss ratio that decreased enough to offset the increase in the average cycles per transfer. The result was bus utilization figures that decreased over most of the block size spectrum (see Figure 5-13). TOPOPT and VERIFY are programs with a fair amount of fine-grain sharing. The resulting increase in their miss ratios (or a very small decrease for some block sizes for VERIFY), plus the normal rise in the average number of cycles per transfer, produced increasing bus utilization

figures (again, see Figure 5-13). ([Cher88] has also noticed the effect of fine-grain sharing on bus traffic. In simulations done on a four-processor multiprocessor, in which management of the 256K byte cache was done under software control, two traces exhibited an increase in bus operations per reference, as block size was increased.)

For three of the traces sharing-related bus overhead comprised a substantial portion of total bus cycles across all block sizes but one (4 bytes for VERIFY). For the fourth trace, TOPOPT, they totally dominated bus activity. The ranges are 56 to 94 percent for TOPOPT, 45 to 61 percent for VERIFY (excluding the exception), 47 to 54 percent for CELL, and 40 to 44 percent for SPICE (see Figure 5-14). (The proportions are higher than the proportions of invalidation misses to total misses, because the cycle figures include cycles for invalidation signals as well as invalidation misses.) The curves clearly show that for 128K byte caches bus bandwidth requirements are determined by the sharing traffic.

Because sharing-related bus overhead is such a large proportion of total bus cycles, its behavior as block size increases can dictate the bus utilization trend. TOPOPT is the most extreme example. It has the largest proportion of sharing cycles, and their rate of increase is steep (see Figure 5-15). Although the uniprocessor cycles decline with increasing block size, their rate of decline is more moderate, and they are a very small proportion of total bus cycles. Therefore TOPOPT's total bus utilization curve rises. The other three traces exhibit similar effects, although for the programs with sequential sharing, the sharing cycle trends pull total bus utilization downwards. (An example appears in Figure 5-16.)

5.3.2. Varying Cache Size

Increasing cache size is an important design technique for improving bus utilization. With the exception of enlarging either an extremely small block or a very large cache,[8] it pro-

[8] Increasing an already large cache is an exception because it provides little additional benefit; on the other hand, doubling a very small block, say 4 bytes in size, produces a good performance improvement.

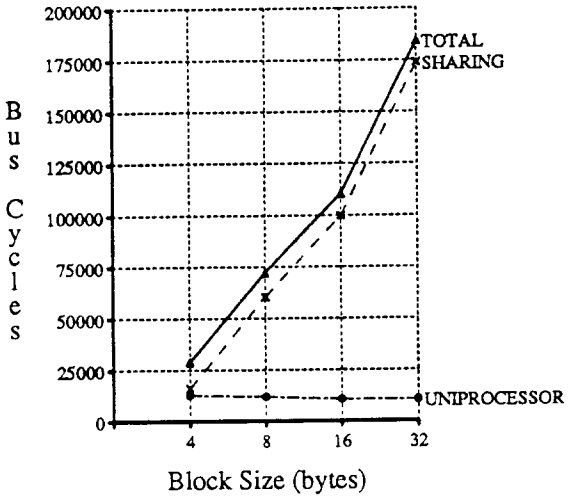

Figure 5-15: Classification of Bus Cycles for TOPOPT

The proportion of sharing-related bus cycles for TOPOPT ranged from 56 to 94 percent. Because they were such a majority of total bus cycles, their behavior forced bus utilization to follow suit.

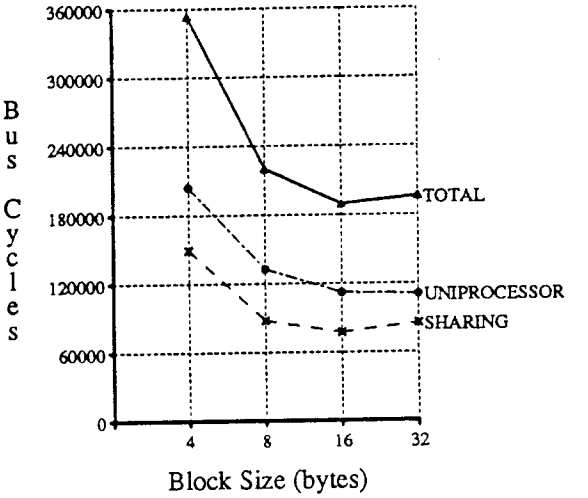

Figure 5-16: Classification of Bus Cycles for SPICE

SPICE was typical of programs with sequential sharing. The decline in sharing-related bus cycles reinforced a corresponding drop in uniprocessor bus cycles, producing a falling bus utilization curve.

vides a larger performance boost than increasing either block size or set associativity [Przy88]. There are two factors that contribute to the greater improvement. First, the miss ratio is more responsive to cache size than to increases in the other two parameters. Second, the longer cache access time of larger caches is less severe a penalty to effective access time than the cost of increasing either of the other parameters, particularly, the increase in bus traffic with a larger block size.

All of the traces exhibit the expected falling bus utilization (see Figure 5-17), and for the usual reason: a miss ratio that declines with increasing cache size (see Figure 5-9). The decline is particularly sharp for the programs with sequential sharing, CELL and SPICE, and their bus utilization curves reflect the drop. The decrease in the miss ratios did not translate directly into

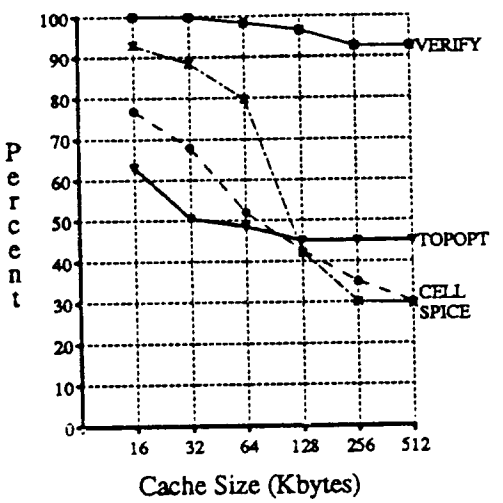

Figure 5-17: Effect of Cache Size on Bus Utilization

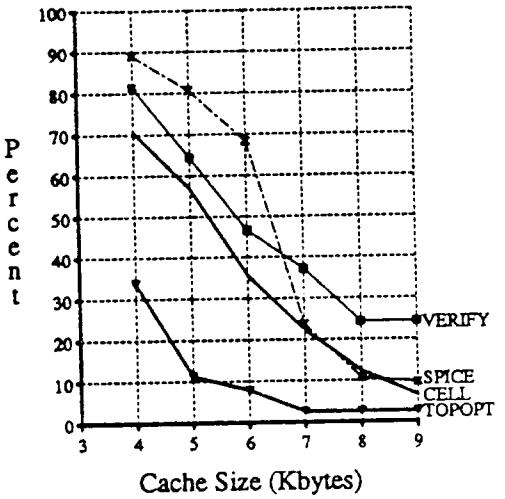

Figure 5-18: Uniprocessor Bus Utilization

As is true for uniprocessor programs, bus utilization for the parallel programs declined with increasing cache size. The benefit of enlarging the cache was greatest for the two programs with sequential sharing, CELL and SPICE. (All cache size graphs assume a 32 byte block.)

Bus utilization was higher than its uniprocessor component. The ranges for the individual traces are: 1.04 to 3.1 higher for SPICE, 1.2 to 3.7 higher for VERIFY, 1.1 to 5.1 higher for CELL and 1.9 to 17.7 higher for TOPOPT.

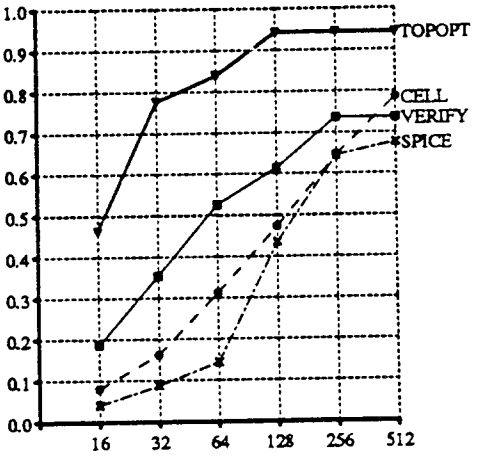

Figure 5-19: Ratio of Sharing Bus Cycles to Total Bus Cycles

The proportion of sharing-related bus cycles to total cycles rises sharply with increasing cache size. For large caches, they comprise the largest component of bus utilization.

a comparable change in bus utilization, because of a rise in both the number of cache-to-cache transfers and the number of invalidations. Under the Berkeley Ownership cache coherency protocol, cache-to-cache transfers are the mechanism for satisfying processor reads to dirty shared data. As cache size increases, the number of cached dirty shared blocks also increases, and therefore the number of cache-to-cache transfers goes up. In the simulator's memory system (and the implementation of SPUR as well), cache-to-cache transfers require more cycles than memory transfers. The shift to the more expensive type of data transfer, as cache size increases, flattens the bus utilization curve. A more optimized cache controller implementation or a slower memory would have produced a steeper drop. (The effect of invalidation signals on bus utilization was discussed in section 5.3.1.)

The proportion of sharing-related bus cycles to total bus cycles is depicted in Figure 5-19. For all traces, cycles due to invalidation signals and invalidation misses rise sharply with cache size. For large caches (128K bytes and up), they dominate bus bandwidth demands. (Results in [Site88] also indicate a rising proportion of sharing traffic with increasing cache size, although the sharing traffic does not dominate, even with one megabyte caches. Traces for their study are concatenated samples of memory references of CAD and expert systems applications running under MACH, in a two processor multiprocessor.)

5.4. Concluding Discussion

5.4.1. Implications for Cache and Bus Designers

Cache design is an optimization problem. Its goal is to minimize effective access time by changing various cache parameters. The difficulty is that these parameters alter cache performance in conflicting ways. For example, increasing cache size decreases the miss ratio, but at the expense of a longer cache access time. Increasing the block size also decreases the miss ratio, but only until the pollution point is reached. After that, larger block sizes produce a rising

miss ratio. An additional drawback of all block size increases is the accompanying increase in the amount of data that is transferred in a single bus operation. The increase in the average cycles per transfer can cause bus utilization to rise even before the pollution point is reached.

Parallel programs, running under write-invalidate coherency protocols, complicate cache design by introducing another factor into the optimization problem: invalidation misses. The studies in this chapter have shown that invalidation misses increase miss ratios, sometimes enough to reverse declining miss ratio curves produced by the other factors. For example, as cache size increases, the number of invalidation misses also increases. Invalidation misses occur in smaller caches as well, but in the guise of replacement misses. With larger caches, some replacement misses for instructions and private data are eliminated; those to shared data can only be converted to invalidation misses. The result is a miss ratio that, for most of the traces, ranges from 2.2 to 4.7 times greater than its uniprocessor component, and 15 times greater in the worst case.

Sharing references also derive less benefit than uniprocessor references from a larger block size. Increasing block size either increases the number of invalidation misses or decreases them at a rate that is less than for uniprocessor misses. The type of miss behavior depends on whether the program exhibits sequential or fine-grain sharing. In the former, invalidation misses decline with block size, and produce a miss ratio that is higher than for comparable uniprocessor programs. When there is fine-grain sharing, the number of invalidation misses rises dramatically with block size. The increase is enough to reverse the declining miss ratio that occurs with uniprocessor programs in caches of this size (128K bytes).

In all cases the miss ratio is higher than in uniprocessor caches. Therefore designers must use larger or more complex caches[9] to obtain the same performance in multiprocessors; even then, they might not be able to obtain this level, because some costs of sharing are inherent in

[9] For example, greater associativity, multi-level caches, etc.

the algorithm, and are unaffected by cache design changes. The choice of block size is dependent on the anticipated workload mix, in particular the balance between programs that exhibit sequential or fine-grain sharing.

The additional cache misses, of course, increase bus utilization. Moreover, sharing under write-invalidate protocols introduces another type of bus operation, the invalidation signal, which further increases bus utilization. Bus utilization was 1.04 to 17.7 times higher with increasing cache size, and 1.3 to 17.7 times higher with increasing block size. Even for the small-scale multiprocessors studied, the bus was well utilized, with typical bus utilization figures ranging from 30 to 70 percent. The implication for bus design is a need for additional speed in order to support a larger scale, single-bus multiprocessor. Fast bus architectures (for example, split transaction bus protocols) and faster bus implementations (for example, bipolar or optics) are even more important in multiprocessors than uniprocessor systems.

5.4.2. Implications for Parallel Software Writers

The performance of parallel programs may be improved by a variety of software techniques for restructuring shared data. The techniques can be used by applications programmers and operating system designers, or compiler writers.

We have seen that shared references were responsible for considerable overhead in the cache and bus performance of parallel programs. Invalidation misses comprised a substantial proportion of total misses for moderate block sizes (32 bytes, and even smaller for some traces) and large cache sizes (128K bytes and up). For all block sizes and large caches, sharing-related bus traffic accounted for the majority of total bus cycles.

As multiprocessor caches continue to increase in size, uniprocessor misses will become a decreasingly smaller proportion of total traffic; and a correspondingly larger proportion will be due to sharing. Adding processors to such systems will increase sharing traffic in absolute terms. The bottom line is that it is the sharing traffic that will determine bus bandwidth

demands, and will eventually limit the scale of the single-bus multiprocessor by creating a bus bottleneck.

Given that multiprocessors already have large caches, the bottleneck can only be postponed by improving the cache and bus performance for the shared data portions of the parallel programs. One observation of the programs studied here is that their memory reference pattern to shared data within the cache block largely determines the coherency cost, measured by miss ratios and bus utilization. Sequential sharing reduces the number of invalidation signals and invalidation misses, which lowers these metrics. On the other hand, fine-grain sharing, i.e., poor sequential sharing, has the opposite effect. Thus better memory organization for shared data can improve program execution.[10] If shared data accessed by different processors are allocated to separate cache blocks, then programs with fine-grain sharing should obtain lower coherency costs, and an improvement in overall performance.

Better data alignment can occur by at least two different means. The first is through explicit programmer specification of the organization of shared data and runtime support for its allocation in shared memory on cache block boundaries. Currently, shared variables may be dynamically allocated by a system runtime routine that makes the data visible to all processes. In the proposed data alignment scheme, the programmer would be responsible for grouping those shared variables that are used by different processors via separate system calls. The routine itself would allocate the shared data in each invocation on cache block boundaries, padding out the block when necessary. The advantage of this approach is the simplicity of its implementation; it is a very straightforward technique for reducing bus traffic under software control. Its disadvantages are that it places the responsibility for optimal runtime memory usage of

[10] Improvements can also come from algorithmic development. For example, waveform relaxation techniques for circuit simulation have better parallel program throughput than the original direct method. The improvement comes because the shared structures (subcircuits) can be more easily partitioned among the processors than in the original method. A further benefit (that also exists in uniprocessor implementations) occurs because only those nodes whose input values have changed during the current time step by a nontrivial amount are reevaluated in the next step.

shared variables entirely on the programmer and requires that the runtime system be aware of the cache block size.

A second method for improving the memory organization of shared data addresses the issue of programmer responsibility, but at an extremely high cost in implementation complexity. The approach involves the automatic compiler detection and consequent memory allocation of per processor shared variables. The techniques involved are similar to those used both for the lifetime analysis of objects to reduce garbage collection overhead [Rugg88] and in restructuring Lisp programs for concurrent execution [Laru88]. The problem is difficult, because the compiler must analyze references to pointers rather than discrete variables. The set of objects that are linked by pointers may be arbitrarily complex, and it is difficult to detect their dynamic relationship. A precise solution is intractable; in practice, the technique could probably only be used for a subset of easily recognizable structures. Moreover, a compile time analysis produces a conservative, worst-case estimate that may not reflect the actual execution behavior of the program. This can lead to wasted memory and additional bus traffic, because small objects would be allocated to larger cache block units. At this point, automatic compiler detection of shared data that is actually used by a single processor is an open research question; it is not clear that freeing the programmer of the responsibility for optimally allocating shared data is worth the complexity of the automatic solution. The programmer-initiated solution should be tried first to determine whether it can produce the performance benefits of good sequential sharing.

5.5. References

[Agar88] A. Agarwal, J. Hennessy and M. Horowitz, "Cache Performance of Operation System and Multiprogramming Workloads", *ACM Transactions on Computer Systems*, 6, 4 (November 1988), 393-431.

[Alex86] C. Alexander, W. Keshlear, F. Cooper and F. Briggs, "Cache Memory Performance in a UNIX Environment", *Computer Architecture News*, 14, 3 (June 1986), 14-70.

[Cher88] D. F. Cheriton, A. Gupta, P. D. Boyle and H. A. Goosen, "The VMP Multiprocessor: Initial Experience, Refinements and Performance Evaluation", *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu, HA (May 1988), 410-421.

[Egge88] S. J. Eggers and R. H. Katz, "A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation", *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu HA (May 1988), 373-383.

[Good87] J. R. Goodman, "Cache Memory Optimization to Reduce Processor/Memory Traffic", *Journal of VLSI and Computer Systems*, 2, 1 & 2 (1987), 61-86.

[Hill87] M. D. Hill, "Aspects of Cache Memory and Instruction Buffer Performance", Technical Report No. UCB/Computer Science Dpt. 87/381, University of California, Berkeley (November 1987).

[Laru88] J. R. Larus and P. N. Hilfinger, "Restructuring Lisp Programs for Concurrent Execution", *Proceedings of the ACM/SIGPLAN Notices PPEALS 1988*, 23, 9 (September 1988), 100-110.

[Przy88] S. Przybylski, M. Horowitz and J. Hennessy, "Performance Tradeoffs in Cache Design", *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu, HA (May 1988), 290-298.

[Rugg88] C. Ruggieri and T. P. Murtagh, "Lifetime Analysis of Dynamically Allocated Objects", *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, San Diego CA (January 1988), 285-293.

[Site88] R. L. Sites and A. Agarwal, "Multiprocessor Cache Analysis Using ATUM", *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu, HA (May 1988), 186-195 .

[Smit85] A. J. Smith, "Cache Evaluation and the Impact of Workload Choice", *Proceedings of the 12th Annual International Symposium on Computer Architecture*, 13, 3 (June 1985), 64-73.

[Smit87] A. J. Smith, "Line (Block) Size Choice for CPU Caches", *IEEE Trans. on Computers*, C-36, 9 (September 1987), 1063-1075.

6 Evaluating the Performance of Four Snooping Cache Coherency Protocols

6.1. Introduction

Both write-invalidate and write-broadcast coherency protocols have been criticized[1] for being unable to achieve good bus performance across all cache configurations. Write-invalidate performance can suffer as coherency block size increases, because of inter-processor contention for addresses within the cache block (see Chapter 5, sections 2.1 and 3.1 and [Egge89]). Large cache sizes will hurt write-broadcast, because of continued bus updates to data that remains in the cache but is no longer actively shared.

Enhancements to the original protocols have been proposed to solve each problem. A read-broadcast extension [Good88, Sega84] to write-invalidate reduces the number of misses for invalidated data by allowing all caches with invalidated blocks to receive new data when any of them issues a read request. It should therefore improve both the miss ratio and bus utilization of

[1] The criticism is unpublished, but widely verbalized in the research community.

write-invalidate. A competitive snooping protocol, introduced in [Karl86, Karl88], was designed to limit the number of broadcasts in write-broadcast. It therefore puts a cap on the performance loss caused by large caches.

The goal of this chapter is twofold: first, to measure the performance problems in the write-invalidate and write-broadcast protocols, as block or cache size increases; and second, to gauge the extent to which the read-broadcast and competitive snooping extensions solve each problem. The results indicate that read-broadcast reduces the number of invalidation misses, but at a high cost in processor lockout from the cache. The net effect can be an *increase* in total execution cycles. Competitive snooping benefits only those programs that exhibit sequential sharing. For programs characterized by inter-processor contention (fine-grain sharing) for shared addresses, competitive snooping can degrade performance by causing a slight increase in bus utilization and total execution time.

The remainder of this chapter contains the two companion protocol studies. Each begins with empirical evidence of the performance loss caused by increasing block or cache size in the original protocol. Then the protocol extensions are described, and the extent to which they improve performance is measured. Section 6.1 briefly reviews two aspects of write-invalidate protocols: the origins of additional coherency overhead caused by large coherency block sizes that was explained in Chapter 4, section 7; and the effects of the overhead on miss ratio and bus utilization studied in Chapter 5. Section 6.2 presents the read-broadcast extension and its benefits and costs to both performance and cache controller implementation. The effects of increasing cache size on bus traffic under write-broadcast protocols is covered in section 6.3. Section 6.4 discusses the competitive snooping alternative and its performance relative to write-broadcast. The last section integrates the results of both studies.

6.2. The Write-Invalidate Protocols

6.2.1. The Write-Invalidate Trouble Spot

Write-invalidate protocols maintain coherency by requiring a writing processor to invalidate all other cached copies of the data before updating its own. It can then perform the current update, and any subsequent updates (provided there are no intervening accesses by other processors) without either violating coherency or further utilizing the bus. Because they create a data writer that can access a shared block without using the bus, write-invalidate protocols minimize the overhead of maintaining cache coherency in two cases: when there are multiple consecutive writes to a block by a single processor (sequential sharing), and when there is little inter-processor contention (fine-grain sharing) for the shared data. Periods of severe contention, however, cause coherency overhead to rise. Inter-processor contention for an address produces more invalidations; the invalidations interrupt all processors' use of the data and increase the number of invalidation misses to get it back. The result is that shared data pingpongs among the caches, with each processor's references causing additional coherency-related bus operations. The greater the number of processors contending for an address, the more frequent the pingponging.

The problem is exacerbated by a large block size, because contention can occur for *any* of the addresses in the block. Therefore the probability that the block will be actively shared increases. An invalidation to one word in a block causes all other words to be invalidated. When other processors subsequently reread these addresses, additional read misses are incurred. The overhead is paid even when a processor reads an address that was not updated. With small block sizes, particularly those of only one word, a write to one address has less effect on reads to another.

6.2.2. Empirical Evidence for the Trouble Spot Analysis

Chapter 5 studied the effect on both miss ratio and bus utilization of increasing block size and cache size under write-invalidate protocols. (The particular write-invalidate protocol used in the simulations was Berkeley Ownership.) The results quantify the loss in performance due to invalidations and invalidation misses. In particular, they support the above analysis concerning the adverse effects of fine-grain sharing, as block size increases.

Parallel programs, with or without contention, suffer from coherency overhead. Unlike uniprocessor misses [Agar88, Alex86, Good87, Hill87, Smit87], invalidation misses react less favorably to increasing block size. Chapter 5 found that the proportion of invalidation misses to total misses actually increased with larger block sizes, and for three of the traces was significant. (The proportions grew from .32 to .37 for CELL, .14 to .30 for SPICE, .06 to .51 for VERIFY and .39 to .94 for TOPOPT, as block size was increased from 4 to 32 bytes.) For programs with sequential sharing (CELL and SPICE), (total) miss ratios were higher than for comparable uniprocessor programs and declined with increasing block size at a slower rate.

The effect on programs with fine-grain sharing (TOPOPT and VERIFY) is more severe. Here invalidation misses increased with increasing block size, not only in proportion to total misses, but in absolute numbers as well. (The proportion of invalidation misses for TOPOPT and VERIFY is stated above; the percentage increase in number of misses was 511 and 840 percent, respectively.) Their dominance was so complete that they reversed the declining miss ratio curves that normally occur with uniprocessor programs in caches of this size (128K bytes).

The additional cache misses increased bus utilization. Moreover, sharing under write-invalidate protocols introduces another type of bus operation, the invalidation signal, which further increased bus utilization. Bus utilization rose 407 and 94 percent for TOPOPT and VERIFY, as block size increased from 4 to 32 bytes. Even for the small-scale multiprocessors studied (12 processors at most), the bus was well utilized, with bus utilization figures of 45 and

97 percent, respectively, at the 32 byte block size. Bus utilization for CELL and SPICE was midrange, higher than for uniprocessor programs, and declined over the block size spectrum.

6.3. The Read-Broadcast Extension

6.3.1. Protocol Description

Since invalidation misses play such a large role in the cache and bus performance of parallel programs at large block sizes, coherency protocols that can reduce them are desirable. *Read-broadcast* [Good88, Sega84] is an enhancement to write-invalidate protocols designed explicitly for this purpose. Under read-broadcast snoops update an invalidated block with data from the bus, whenever they detect a read bus operation for the block's address. Detection is positive whenever the tag of the snooped address matches that of a cached block, and the block state is invalid.

The read-broadcast extension adds little complexity to the cache controller hardware. An examination of the SPUR cache controller implementation indicates that one additional minterm is required in the snoop PLA for the detection. Assuming that the snoop can have access to the cache in a short and bounded amount of time, a buffer large enough to hold the data as it comes from the bus is also needed. If timely snoop access to the cache cannot be guaranteed, an extra bus line is necessary to delay transmission of the data. Finally, control to implement read-interference[2] is required to meet the invalidation miss limit, described below.

The technique improves the performance of write-invalidate by limiting the number of invalidation misses to one per invalidation signal. One invalidation miss occurs if the bus operation is a read issued by a cache with a previously invalidated block. No invalidation

[2] Read-interference occurs when a processor has queued a bus read request for an address that is read-broadcast before the requesting processor obtains the bus. During the read-broadcast the requesting processor updates its cache with data from the bus. Therefore it can satisfy its read reference directly from the cache and no longer requires the bus operation. Control is needed to detect the interference and cancel the pending read bus operation.

misses result when the bus read is a first-reference or replacement miss. Subsequent rereads by processors that have received data on a read-broadcast will be a cache hits rather than invalidation misses.

6.3.2. Read-Broadcast Results

6.3.2.1. The Benefits to Miss Ratio and Bus Utilization

Read-broadcast reduced the number of invalidation misses (see Table 6-1). For three of

Comparison of Berkeley Ownership & Read-Broadcast							
Trace	Block Size (bytes)	Invalidation Misses			Miss Ratio		
		Berk. Own.	Read Bdcast.	Decrease (percent)	Berk. Own.	Read Bdcast.	Decrease (percent)
CELL	4	22649	13566	40.1	1.93	1.67	13.7
CELL	8	18823	11264	40.2	1.49	1.28	14.1
CELL	16	15040	8942	40.5	1.10	0.93	15.6
CELL	32	11748	7325	37.6	0.86	0.73	14.4
SPICE	4	6918	6663	3.7	2.90	2.97	-2.2
SPICE	8	4143	3870	6.6	1.64	1.65	-0.2
SPICE	16	3607	3447	4.4	1.09	1.10	-0.4
SPICE	32	3726	3009	19.2	0.77	0.74	3.4
TOPOPT	4	1890	922	51.2	0.15	0.12	20.1
TOPOPT	8	6117	4706	23.1	0.25	0.20	17.2
TOPOPT	16	8835	6459	26.9	0.30	0.23	23.2
TOPOPT	32	11556	7385	36.1	0.37	0.25	33.8
VERIFY	4	2441	2062	15.5	1.42	1.41	1.0
VERIFY	8	8921	7786	12.7	1.38	1.34	2.6
VERIFY	16	15371	11497	25.2	1.40	1.28	9.1
VERIFY	32	22957	13717	40.2	1.45	1.17	19.4

Table 6-1: Comparison of Invalidation Misses and Miss Ratio
for Berkeley Ownership and Read-Broadcast

This table depicts the decline in the number of invalidation misses and the miss ratio that occured with read-broadcast. The drop in invalidation misses was less pronounced for SPICE, because its shared data had been optimized for a block size larger than the maximum studied here. This small decline, coupled with a slight rise in uniprocessor misses, produced rising miss ratios (negative decreases) for some block sizes. (All simulations were run with a 128K byte cache; miss ratios are the geometric mean across all processors.)

the traces (CELL, TOPOPT and VERIFY) the drop ranged from 13 to 51 percent, over all block sizes. The decrease for SPICE was much lower. SPICE data structures had been explicitly sized to the ELXSI 6400 64-byte cache block to avoid inter-processor contention for addresses within a block. Therefore, for block sizes considered in this study, up to 32 bytes, little contention was observed; and read-broadcast consequently brought less benefit.

Because of the decrease in invalidation misses, the proportion of invalidation misses within total misses was less than for write-invalidate (see Figures 6-1 and 6-2). This is important, because increases in block and cache size produce steeper reductions in uniprocessor misses than invalidation misses. Therefore, to the extent that misses in parallel programs are caused by normal cache accesses rather than sharing activity, cache performance will improve as block and cache sizes increase. At larger block sizes invalidation misses for CELL, TOPOPT and VERIFY dropped to between a quarter and a third of the total. (Under Berkeley Ownership they had ranged from thirty to over forty percent.) But for TOPOPT invalidation misses still dominated miss ratio behavior at most block sizes (90 percent at 32 bytes at maximum). As with the original write-invalidate protocol, the ratio of invalidation to total misses for all traces rose with increasing block size.

For the most part the consequence of the drop in invalidation misses was a decline in the total miss ratio (again, see Table 6-1). CELL and TOPOPT had moderate decreases (13.7 to 15.6 percent and 17.2 to 33.8 percent, respectively); VERIFY had a wider range of decrease (1.0 to 19.3 percent). The miss ratio for SPICE did not decline across all block sizes, and, when it did, the decrease was small. The small increases occured because the samples in comparative (Berkeley Ownership vs. read-broadcast) simulations covered a slightly different set of references. The difference in samples was caused by the elimination of invalidation misses from the read-broadcast simulations. Changing invalidation misses to cache hits allows processors to process references more quickly than under Berkeley Ownership. The effect is to slightly alter

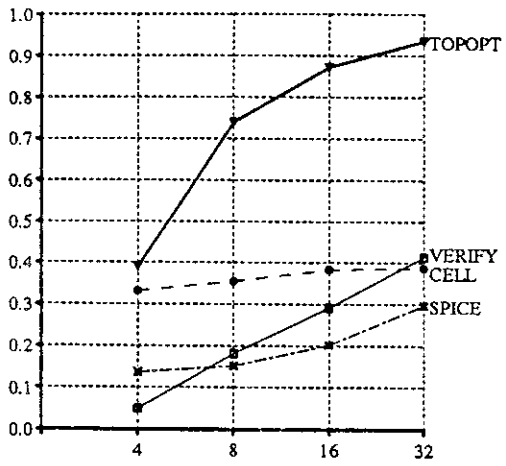

Figure 6-1: Ratio of Invalidation Misses to Total Misses for Berkeley Ownership

The ratio of invalidation misses to total misses increases as block sizes increase. At larger block sizes the invalidation misses of three of the traces comprise a substantial portion of the total; and for TOPOPT they dominate miss ratio behavior. (The numbers are the geometric mean of the ratio of invalidation to total misses, across all processors.)

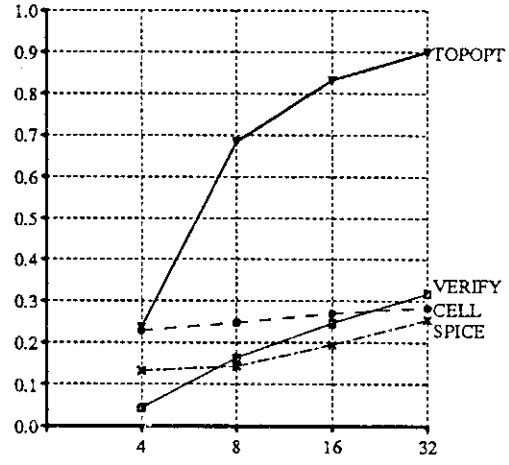

Figure 6-2: Ratio of Invalidation Misses to Total Misses for Read-Broadcast

Under read-broadcast the ratio of invalidation misses to total misses still increases with block size, although the proportions are lower than with Berkeley Ownership. At larger block sizes the invalidation misses for three of the traces have dropped to between a quarter and a third of the total; for TOPOPT they still dominate miss ratio behavior.

the set of references executed and the global order in which they are processed under the two protocols. For SPICE the consequence was a slight rise in the uniprocessor component of the miss ratio for read-broadcast (relative to Berkeley Ownership), which offset the small decline in the number of invalidation misses. For the other traces the sample discrepancy was considerably less, the uniprocessor misses were almost identical, and the reduction in the number of invalidation misses was also greater. Therefore the drop in invalidation misses produced a corresponding decline in the miss ratio.

The critical system bottleneck in a single-bus, shared memory multiprocessor is the bandwidth of the system bus. Therefore the most important consequence of read-broadcast is

the effect of its lower miss ratios on bus utilization. The improvement (i.e., drop in bus utilization) ranged from 8.7 to 10.9 percent for CELL, .8 to 5.1 percent for SPICE, 14.3 to 22.6 percent for TOPOPT and .8 to 11.5 percent for VERIFY. (Details appear in Table 6-2.) To put the read-broadcast benefit in perspective, the change was large enough to allow an additional two processors for TOPOPT, and one each for CELL and VERIFY, and still maintain the same level of bus utilization. (SPICE had lower bus utilization for the block sizes that had a slight rise in the miss ratio, because the total cycles in the simulation were higher with read-broadcast. The cycle increase was due to a greater delay in obtaining the bus and several other read-broadcast-related factors that are discussed below.)

Comparison of Berkeley Ownership & Read-Broadcast				
Trace	Blocksize (bytes)	Bus Utilization		
		Berkeley Ownership	Read Broadcast	Decrease (percent)
CELL	4	42.155	38.470	8.743
CELL	8	39.798	35.849	9.924
CELL	16	38.592	34.383	10.906
CELL	32	42.559	38.042	10.614
SPICE	4	59.546	59.070	0.798
SPICE	8	44.821	44.159	1.477
SPICE	16	40.298	39.948	0.870
SPICE	32	42.221	40.061	5.117
TOPOPT	4	8.925	6.979	21.806
TOPOPT	8	21.289	18.247	14.288
TOPOPT	16	30.972	25.656	17.165
TOPOPT	32	45.108	34.895	22.640
VERIFY	4	49.738	49.346	0.788
VERIFY	8	68.380	66.802	2.307
VERIFY	16	84.760	79.215	6.543
VERIFY	32	96.566	85.491	11.469

Table 6-2: Comparision of Bus Utilization for Berkeley Ownership and Read-Broadcast

This table depicts the decline in bus utilization that occured with read-broadcast over Berkeley Ownership. (All simulations were run with a 128K byte cache; bus utilization figures are the geometric mean across all processors.)

The magnitude of the drop in both miss ratio and bus utilization was moderate. The performance gain was less than expected because of the extremely sequential nature[3] of the sharing in the programs. Sequential sharing can be measured by several metrics (see Chapter 4, section 3.2). The most pertinent for a study of invalidation misses is the average number of processors that reread an address between writes by different processors. For all traces this figure averaged around one (1.1 for CELL, .7 for SPICE, .8 for TOPOPT and 1.0 for VERIFY), with the distribution heavily weighted by zeros and ones. (CELL had the most evenly spread distribution, with 2 or more processors rereading between 25 and 21 percent of the time. This accounts for its greater decline in invalidation misses. SPICE had the most skewed distribution, with between 91 and 98 percent of the writes followed by zero or one rereads. Its improvement was the least of the traces.) In actual practice the number of invalidation misses was quite close to the read-broadcast limit of one. This was true even for the traces characterized by fine-grain sharing (TOPOPT and VERIFY). If there had been more processors involved in the contention, read-broadcast would have provided greater benefit.

6.3.2.2. The Cost in Per Processor and System Throughput

The reduction in invalidation misses did not come for free. Read-broadcast has two side effects that contribute to processor execution time: an increase in processor lockout from the cache[4] and an increase in the average number of cycles per bus transfer. Their consequence for three of the traces was an increase in total execution cycles over the Berkeley Ownership simulations.

The more important of the two factors is the increase in processor lockout from the cache. Cache lockout occurs because of CPU and snoop contention over the shared cache resource.

---

[3] Recall that in sequential sharing each processor completes multiple accesses to the shared data before another processor begins. The alternative is fine-grain sharing, in which there is inter-processor contention for the data.

[4] I am referring to the data RAMs. As stated in Chapter 3, there are two copies of the tags and state, one for the CPU and one for the snoop.

The CPU must use the cache for fetching the current instruction (on a miss in the on-chip instruction cache or for all instructions if there is no on-chip cache), obtaining data referenced by the current instruction, and prefetching subsequent instructions. In machines like the one being simulated, with a RISC-based architecture, no on-chip instruction cache and a cache access time that matches the cycle time of the CPU, the CPU needs to access the cache each cycle.[5] At the same time, the snoop also needs access to the cache for maintaining coherency. Read-broadcast requires more snoop-related cache activity than Berkeley Ownership, because snoops must deposit data into the cache on some bus reads and more snoops must update the processor's cache state on subsequent invalidations. The first operation does not occur under Berkeley Ownership, and the latter occurs less frequently. Both activities divert the CPU from its normal instruction execution and contribute to program slowdown.

The increase in lockout with read-broadcast was substantial (278 to 305 percent for CELL, 147 to 191 percent for SPICE, 35 to 87 percent for TOPOPT and 143 to 329 percent for VERIFY). On the average 42 percent of total lockout cycles was attributable to taking data on read-broadcasts, and 40 percent to the state updates. (Cache-to-cache transfers account for the remainder.) The increase due to these factors was softened somewhat by the lockout savings from a decline in cache-to-cache transfers that had satisfied invalidation misses under Berkeley Ownership.

However, in terms of total execution cycles, processor lockout was a minor cost. The ratio of lockout to total cycles averaged 5.8 percent for all traces, across most block sizes. The lone exception was VERIFY's 32 byte block simulation, in which processor lockout accounted for an appalling 21 percent of total cycles. The importance of processor lockout is that for three of the traces (CELL, SPICE and VERIFY), its increase *wiped out the benefit to total execution cycles gained by the decrease in invalidation misses.* The consequence was a slight increase in

[5] In CPUs with instruction caches on-chip, prefetching accesses would replace many of the instruction accesses.

total execution cycles, ranging from .9 to 3.6 percent. The lone exception was TOPOPT, in which the benefit from declining invalidation misses was greater than the cost of processor lockout; here the improvement in total execution cycles varied from .1 to 7.7 percent, as block size increased from 4 to 32 bytes.

The negative effect of processor lockout would not be as severe with a more optimized cache controller implementation. In the SPUR implementation, the priority for using the cache belongs to the processor rather than the snoop, and the two run on asynchronous clocks. Therefore the snoop must negotiate to obtain use of the cache (via separate request and grant cycles), and acknowledge that it has finished. A more optimized implementation would eliminate the handshaking cycles by using a single clock for the entire system.

A lower bound can be placed on processor lockout by eliminating the extra cycles from the above results: read-broadcast is then assumed to cost only the number of cycles needed to fill the cache. The results indicate that, even under these best case assumptions, the increase in processor lockout cycles is greater than the decrease in invalidation miss cycles for more than half the simulations. For these simulations read-broadcast still causes a net gain in total execution cycles. (The major exception was TOPOPT. Since it had fewer execution cycles under read-broadcast even with the less optimized implementation, it is not surprising that the lower bound assumptions would bring further improvement.)

The second factor that contributed to an increase in processor execution time was a rise in the average number of cycles per bus transaction. The increases ranged from .3 to 3.1 percent, for all traces and over all block sizes, and averaged around one. There are two causes. The first is the additional cycle required in the read-broadcast implementation for the snoops to acknowledge that they have completed the operation. Under write-invalidate the same snoops are not actively involved in the bus operation; they merely do a lookup and decide to take no action. The lookup can easily be subsumed in the time required for either the cache-to-cache or

memory transfer that satisfies the invalidation miss. The second is the need to update the processor's state on both read-broadcasts and simple state invalidations. For both operations more caches are involved than with invalidation misses and state invalidations under Berkeley Ownership. Therefore there is a greater probability that the update will be delayed, because the processor is using the cache to service a memory request.

6.3.3. Write-Invalidate/Read-Broadcast Summary

The criticism of write-invalidate, that multiple-processor contention within the block would cause excessive invalidation misses as block size is increased, was not born out by the analysis of these traces. It is true that the number of invalidation misses rose with increasing block size, and for the traces with fine-grain sharing this caused an adverse effect on miss ratios and bus utilization. However, most of these misses were caused by a reread by a *single* processor. Therefore the read-broadcast solution had less impact than was originally postulated.

Still, at first glance it appears that read-broadcast is a good extension to the write-invalidate protocols, primarily because it is an extremely low cost solution for the moderate benefit it provides. However, when the increase in both processor lockout and average cycles per bus transaction are considered, for most of the simulations the result is a net *gain* in total execution cycles.

Read-broadcast would be more beneficial if two conditions were different. The most important is if the workload were one in which more processors were contending for the data (for example a one producer/several consumers situation). In this case the reduction in invalidation misses would be greater. The second condition, which is a second order effect, is a more optimized cache controller implementation, designed to minimize the cycles consumed during processor lockout.

6.4. The Write-Broadcast Protocols

6.4.1. The Write-Broadcast Trouble Spot

Write-broadcast protocols broadcast updates to shared addresses, so that all caches and memory have access to the most current value. Coherency overhead stems entirely from the bus broadcasts. They occur for all updates to data that are contained in more than one cache, and for the first update to an address after the writing processor has the only copy. (In this case the block has been replaced in the other caches.)

Chapter 5 demonstrated that sharing-related bus traffic will require multiprocessors to have larger or more complex caches than uniprocessors to obtain comparable performance. The requirement is particularly troublesome for the write-broadcast protocols, because larger cache sizes can cause an increase in broadcast operations. As cache size grows, the lifetime of cache blocks increases because of a decline in block replacements. Shared data tends to remain in a cache for longer periods of time, long past the point when its processor has finished accessing it. However, its presence in the cache drives the shared bus line, giving the illusion of sharing. Therefore write-broadcasts continue for data that is no longer being actively shared.

6.4.2. Empirical Support for the Trouble Spot

The traces confirm this analysis. For all traces, the number of write-broadcasts rises with increasing cache size (see Figure 6-3). CELL and SPICE have a much larger increase than TOPOPT and VERIFY (84.2 and 100.3 percent over the entire cache size range, versus 3.7 and 15.2 percent). The steepness of their rise correlates with several factors, the most important of which is the pattern of inter-processor references to shared data. For CELL and SPICE this pattern is characterized by sequential sharing for shared data in a coherency block. Sequential sharing is indicated by long average write run lengths for the blocks. (The exact figures are 4.9 writes per write run for CELL and 6.2 for SPICE.) In small caches not all the writes in a long

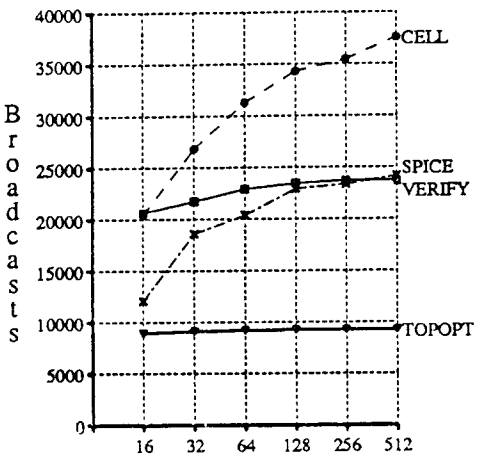

Figure 6-3: Write Broadcasts to Shared Data under Firefly

In the Firefly protocol the number of write-broadcasts increases with increasing cache size for all traces, given credence to the "illusion of sharing" hypothesis.

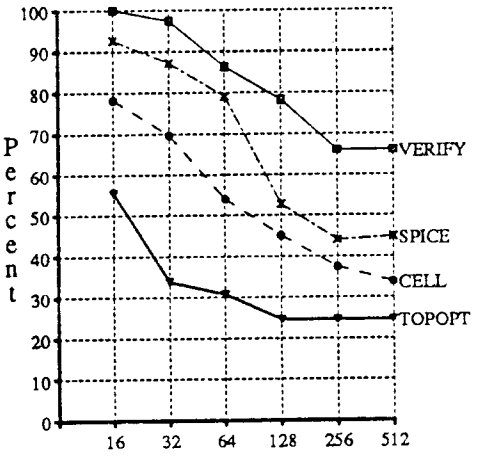

Figure 6-4: Bus Utilization under Firefly

Despite the rise in write-broadcasts, bus utilization fell because of the benefits of large caches on uniprocessor misses.

write run result in write-broadcasts. First, shared data is replaced more frequently than in larger caches, and, secondly, in these traces only two processors are involved in the sharing the vast majority of the time. The combined effect is that data may reside in only one cache for the final writes in a write run, allowing these writes to take place locally. In an infinite cache, *all* writes become write-broadcasts, because blocks remain in the cache indefinitely. Therefore, as cache size increases, more writes in a long write run will result in bus broadcasts; and the greater the average write run length, the greater the increase in write-broadcasts. TOPOPT and VERIFY, on the other hand, had short average write run lengths, 1.21 and 2.2, respectively. The smaller length was one of the factors responsible for the more level write broadcast curves, as cache size increased.

A second factor contributing to the shape of the curves is the rate of block replacement. Within a particular trace, the increase in write-broadcasts (with cache size) is most pronounced for smaller caches, where the drop in block replacements is also greatest. Finally, at large cache sizes the working sets of TOPOPT and VERIFY fit into the cache. The number of block replacements drops to zero and the level of write-broadcasts remains constant.

Despite the rise in write-broadcasts, bus utilization fell for all traces (see Figure 6-4).[6] The decrease is due to the positive effects of increasing cache size on the uniprocessor component of bus utilization, which dropped an average of 84 percent over the cache size range. It is offset somewhat by the increase in write-broadcast cycles (see a representative trace in Figure 6-5).

For all traces, the proportion of write-broadcast cycles within total cycles increased dramatically with increasing cache size (see Figure 6-6). The increase only leveled off at the point at which the working set of the program fit into the cache. At the largest cache sizes the write-broadcast cycles dominated bus activity for all traces. The high ratio of sharing cycles to total cycles means that with large cache sizes, sharing bus traffic will be the cause of the bus bottleneck. Therefore a protocol that limits the number of write-broadcasts is desirable.

6.5. Competitive Snooping

6.5.1. Protocol Description

Competitive snooping [Karl86, Karl88] is a write-broadcast protocol that switches to write-invalidate when the breakeven point in bus-related coherency overhead between the two approaches is reached. The breakeven point for a particular address occurs when the sum of the write broadcast cycles issued for the address equals the number of cycles that would be needed for rereading the data had it been invalidated. Competitive snooping thus limits coherency

[6] The only exception is the transition to a 512K byte cache for SPICE.

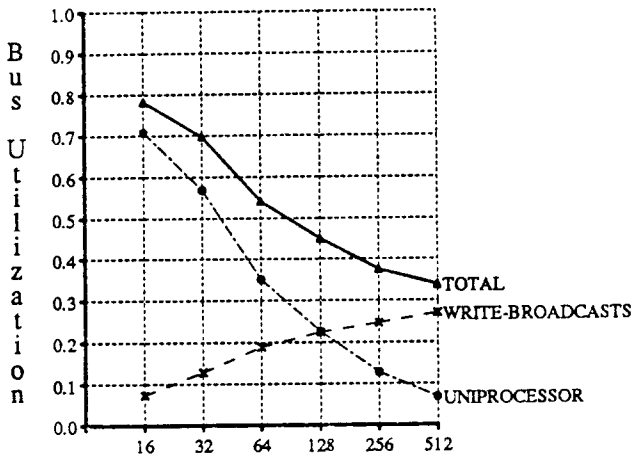

Figure 6-5: Bus Cycles for CELL under Firefly

This classification of bus cycles for CELL illustrates the effect of write-broadcast cycles on total bus cycles, using the Firefly protocol. Write-broadcast cycles rise with increasing cache size; uniprocessor bus cycles tend to fall. The two effects produce bus utilization that still declines, but less steeply than for uniprocessor programs.

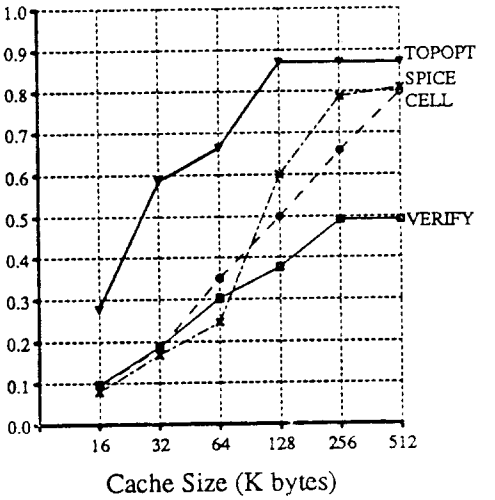

Figure 6-6: Ratio of Broadcast Cycles to Total Bus Cycles

The ratio of write-broadcast cycles to total bus cycles increases with increasing cache size under Firefly. The rise is much steeper for the traces with longer average write run lengths, CELL and SPICE.

overhead to twice that of optimal.[7]

The first algorithm proposed in [Karl86] (called "Standard-Snoopy-Caching") assumes that an adversary can choose any processor to either write or reread a shared address. A counter, whose initial value is the cost in cycles of a data transfer, is assigned to each cache block in every cache. On a write broadcast, a cache that contains the address of the broadcast is

[7] Larry Rudolph makes a very apt analogy between the rationale behind competitive snooping and the dilemma faced by any novice skier. The beginning skier is hesitant to buy skis immediately for fear that his/her interest in skiing might be a passing fancy. On the other hand renting week after week can be costly. The pivotal question is therefore *when* to stop renting and make the purchase. Not knowing ahead of time which will be his or her preference, the budding skier should rent until he or she has spent an amount equivalent to the purchase price of new skis; and then buy the skis. Like competitive snooping, this course of action limits the total cost to twice that of optimal.

(arbitrarily)[8] chosen, and its counter is decremented. When a counter value reaches zero, the cache block is invalidated. When all counters for an address, other than that of the writer, are zero, write-broadcasts for it cease. Any reaccess by a processor to an address resets its cache's counter to the initial value. The algorithm's lower bound proof demonstrates that the total costs of invalidating are in balance with the total costs of rereading.

In an alternate algorithm (called "Snoopy-Reading") the adversary is allowed to read-broadcast on rereads. In order to obtain the lower bound of the previous algorithm, the coherency algorithm is given the same capability. All other caches with invalidated copies take the data, and reset their counters. As in the original scheme, when a cache's counter reaches zero, it invalidates the block containing the address; and write broadcasts are discontinued, when all caches but that of the writer have been invalidated.

Read-broadcasting by the adversary also prompts other changes in the coherency algorithm. For example, on a write-broadcast *all* caches that contain the updated address decrement their counters rather than only one; and the decrementing is done on consecutive write broadcasts *by a particular processor*, rather than any processor. The simultaneous decrements complement the simultaneous cache updates on read-broadcasts, i.e., they reduce the costs of broadcasting to match the cheaper rereads. The single writer requirement corresponds to all counters being reset on an access by another processor. More than one processor referencing the data indicates (obviously) that there is sharing. As long as data is shared, a good competitive coherency algorithm will broadcast rather than invalidate. Broadcasting occurs as long as counter values are greater than zero. Therefore when a processor other than the writer accesses the data, all counters are reset to force broadcasting.

The advantages of the alternate scheme over the original are that (1) it is well suited for a workload in which there are few rereads (as is the case with these traces) and (2) its

[8] The particular choice of cache does not affect the worst-case bound.

implementation doesn't require hardware to "arbitrarily" choose a cache for counter decrementing. When there are few rereads, a competitive coherency algorithm should make the data private sooner rather than later, in order to avoid unnecessary broadcasts. By requiring all processors to decrement their counters simultaneously, Snoopy-Reading can invalidate more quickly than Standard-Snoopy-Caching.

In the simulator's implementation of Snoopy-Reading, a writing processor keeps track of the number of its consecutive writes to each address (through cache state values). When the breakeven point for broadcasts has been reached, it signals to the other caches to invalidate. The breakeven point was defined to be the maximum of the ratio of data transfer to write-broadcast cycles that is used in the algorithm and the value three. The constant insures that write-broadcasts will continue long enough to prevent busywaiting over the bus. A processor uses the first of the three broadcasts for setting the lock, and the second for clearing it. At this point the lock is still present in other caches, and processors can detect locally that it has been freed. On the third broadcast (which, if it occurs, demonstrates that the address is not a lock), the data is invalidated. This implementation requires a six-value coherency state, and a correspondingly larger PLA for both the snoop and the portion of the cache controller that services memory requests for the CPU.

6.5.2. Competitive Snooping Results

Competitive snooping decreased the number of write-broadcasts issued for all traces (see Table 6-3). The benefit was greater for those traces whose pattern of access to shared data within a coherency block was characterized by sequential sharing (CELL and SPICE). Recall that their average write run lengths were 4.9 and 6.2. Given the breakeven point in the simulations, each trace saved on the average, 2 or 3 broadcasts each time a different processor wrote to

Write-Broadcasts				
Trace	Cache Size (Kbytes)	Firefly	Competitive Snooping	Percentage Change
CELL	16	20402	13199	35.31
CELL	32	26841	15507	42.23
CELL	64	31300	15514	50.43
CELL	128	34287	15212	55.63
CELL	256	35444	15192	57.14
CELL	512	37579	15338	59.18
SPICE	16	12076	4510	62.65
SPICE	32	18555	5900	68.20
SPICE	64	20362	6373	68.70
SPICE	128	22925	7045	69.27
SPICE	256	23344	7251	68.94
SPICE	512	24184	7412	69.35
TOPOPT	16	8918	8218	7.85
TOPOPT	32	9111	8352	8.33
TOPOPT	64	9190	8410	8.49
TOPOPT	128	9244	8458	8.50
TOPOPT	256	9244	8458	8.50
TOPOPT	512	9244	8458	8.50
VERIFY	16	20589	18091	12.13
VERIFY	32	21726	18835	13.31
VERIFY	64	22914	19097	16.66
VERIFY	128	23476	19107	18.61
VERIFY	256	23719	19330	18.50
VERIFY	512	23719	19330	18.50

Table 6-3: Comparison of Write-Broadcasts for Firefly and Competitive Snooping

This table depicts the decline in the number of write-broadcasts that occured with competitive snooping. The drop was most pronounced for CELL and SPICE, which had the longest average write run lengths. Identical values across cache sizes for TOPOPT and VERIFY indicate that their working sets fit into the caches. (All simulations were run with a 32 byte block.)

a shared address.[9] The average write run lengths for TOPOPT and VERIFY were below the simulator's breakeven point (1.2 and 2.2, respectively). Therefore no broadcast savings was accrued in most cases.

[9] Technically this is true only for the large caches. At smaller cache sizes the savings would be less. See the discussion on the effect of average write run length on write-broadcast protocols in section 6.4.2.

The corresponding decrease in the number of write-broadcast cycles was offset to varying extents by the additional cycles for invalidation signals and invalidation misses (see Table 6-4). For CELL and SPICE the effect was to reduce the percentage improvement in cycles consumed in sharing-related bus operations to 10 to 26 percent for CELL and 49 to 52 percent for SPICE.

Sharing Cycles							
Trace	Cache Size (Kbytes)	Firefly	Competitive Snooping				% Change
		Write Bdcasts.	Write Bdcasts.	Invals.	Inval. Misses	Total	
CELL	16	167122	108850	24489	17820	151159	9.55
CELL	32	221925	129716	33051	28706	191473	13.72
CELL	64	259327	130740	37395	39140	207275	20.07
CELL	128	285430	129361	40597	51286	221244	22.49
CELL	256	295069	129527	41450	55567	226544	23.22
CELL	512	312668	130360	42849	57944	231153	26.07
SPICE	16	102645	39190	7912	2236	49338	51.93
SPICE	32	158119	51491	13660	12786	77937	50.71
SPICE	64	172139	55384	15115	15168	85667	50.23
SPICE	128	191106	60515	18126	18068	96709	49.40
SPICE	256	193971	61880	18491	18262	98633	49.15
SPICE	512	200782	63020	19076	18907	101003	49.70
TOPOPT	16	75828	74927	1603	2655	79185	-4.43
TOPOPT	32	77214	76249	1916	3366	81531	-5.59
TOPOPT	64	77936	76821	1920	3238	81979	-5.19
TOPOPT	128	78256	77120	1942	3380	82442	-5.35
TOPOPT	256	78256	77120	1942	3380	82442	-5.35
TOPOPT	512	78256	77120	1942	3380	82442	-5.35
VERIFY	16	170952	155223	9228	8679	173130	-1.27
VERIFY	32	183516	165910	10798	12157	188865	-2.91
VERIFY	64	194813	170477	12007	15809	198293	-1.79
VERIFY	128	199733	171116	12744	18125	201985	-1.13
VERIFY	256	200341	171961	13323	19132	204416	-2.03
VERIFY	512	200341	171961	13323	19132	204416	-2.03

Table 6-4: Comparison of Sharing Cycles for Firefly and Competitive Snooping

This table depicts the difference in the number of cycles for the sharing-related bus operations for Firefly and competitive snooping. The decline in write-broadcast cycles is offset by cycles for invalidation signals and invalidation misses. For TOPOPT and VERIFY the combination of a smaller cycle savings in write-broadcasts and the additional cycles related to invalidations produced a net increase in sharing-related cycles. (All simulations were run with a 32 byte block.)

However, the savings was still substantial enough to cause a drop in bus utilization relative to write-broadcast. The decline in bus utilization for CELL ranged as high as 19 percent; for SPICE as high as 30 percent. For all simulations but two (CELL with 16K and 32K byte caches) the lower bus utilization produced fewer total execution cycles.

For TOPOPT and VERIFY the smaller decline in write-broadcasts, coupled with the additional cycles for invalidation signals and invalidation misses, produced an *increase* in sharing-related bus cycles. This increase was responsible for a slight rise in their bus utilization figures over write-broadcast (1.6 to 4.5 percent for TOPOPT and .8 percent at most for VERIFY). Higher bus utilization brought an increase in total execution cycles. (Details on bus utilization and total execution cycles appear in Table 6-5.)

6.5.3. Write-Broadcast/Competitive Snooping Summary

The extent to which competitive snooping improves the performance of write-broadcast depends on the pattern of references to shared data. When sharing is sequential, as exhibited by relatively longer average write run lengths, the benefit is greatest. Here the savings in write-broadcast cycles decreases bus utilization and total execution time. As inter-processor contention for the shared addresses rises, competitive snooping becomes less attractive. The decrease in write-broadcasts diminishes, and in some cases can be offset by the rise in invalidations and the more expensive (in numbers of cycles) invalidation misses. The result is an increase in bus utilization and total execution time. (An alternative argument is that programs with fine-grain-sharing for shared addresses are a good match for write-broadcast protocols. Therefore, they have less need for competitive snooping, and it consequently provides less benefit.)

6.6. Chapter Summary

This chapter contains two companion studies of bus-based, shared memory cache coherency protocols. The purpose of each is twofold: first, to measure the performance loss of

Bus Utilization & Total Execution Cycles							
Trace	Cache Size (Kbytes)	Bus Utilization			Total Execution Cycles		
		Firefly	Compet. Snooping	% Chg.	Firefly	Compet. Snooping	% Chg.
CELL	16	78.21	78.24	-0.04	2251417	2275472	-1.07
CELL	32	69.65	69.41	0.34	1722507	1726670	-0.24
CELL	64	54.07	51.94	3.95	1367997	1358706	0.68
CELL	128	45.13	41.29	8.49	1267754	1246316	1.69
CELL	256	37.52	32.68	12.90	1196530	1170737	2.16
CELL	512	33.88	27.56	18.67	1156534	1128079	2.46
SPICE	16	92.66	92.24	0.46	1385228	1344603	2.93
SPICE	32	87.17	86.09	1.24	1078916	1007891	6.58
SPICE	64	79.10	77.34	2.22	886776	795919	10.25
SPICE	128	52.80	43.43	17.74	603795	517028	14.37
SPICE	256	44.06	31.98	27.42	559356	474377	15.19
SPICE	512	44.88	31.38	30.08	553071	474123	14.27
TOPOPT	16	55.55	56.42	-1.56	491294	495603	-0.88
TOPOPT	32	33.89	34.96	-3.13	389304	391695	-0.61
TOPOPT	64	30.76	31.82	-3.43	381349	382676	-0.35
TOPOPT	128	24.68	25.79	-4.51	364345	364798	-0.12
TOPOPT	256	24.68	25.79	-4.51	364345	364798	-0.12
TOPOPT	512	24.68	25.79	-4.51	364345	364798	-0.12
VERIFY	16	99.97	99.97	0.00	1760674	1786211	-1.45
VERIFY	32	97.41	97.58	-0.17	1002740	1017567	-1.48
VERIFY	64	86.24	86.58	-0.39	744443	749358	-0.66
VERIFY	128	78.18	78.25	-0.08	677634	682098	-0.66
VERIFY	256	65.99	66.08	-0.14	617141	622265	-0.83
VERIFY	512	65.99	66.08	-0.14	617141	622265	-0.83

Table 6-5: Comparison of Bus Utilization & Total Execution Cycles for Firefly and Competitive Snooping

This table depicts the change in the bus utilization and total execution cycles that occured with competitive snooping. The decrease in sharing-related cycles for CELL and SPICE resulted in a decline in both. And, conversely, the increase in sharing cycles for TOPOPT and VERIFY produced a rise. (All simulations were run with a 32 byte block.)

changing particular cache parameter values on well-known snooping coherency techniques; second, to determine to what extent extensions, designed specifically to eliminate deficiencies in the original protocols, achieve performance improvements. In the first study, read-broadcast was proposed to eliminate the rise in invalidation misses in write-invalidate protocols that occur with increasing block size. In the second, competitive snooping was intended to limit the

increase in write-broadcasts caused by increasing cache size in write-broadcast coherency protocols.

The results have found that neither extension produces a savings in coherency overhead across all workloads studied. In those cases in which there was a performance loss, the original protocol, write-invalidate or write-broadcast, was a good match for the program. Therefore there was not much room for improvement; and the extension often introduced secondary costs which outweighed the small savings in coherency overhead. Furthermore, both extensions required some additional hardware complexity.

The workload used in these studies is characterized by sequential sharing, i.e., data is shared by very few processors at a time. Therefore read-broadcast reduced the number of invalidation misses only moderately, and at a high cost in processor lockout from the cache. In some cases, the net effect was an increase in total execution cycles. These results clearly indicate that read-broadcast is inappropriate for programs with sequential sharing. However, if more processors had been involved in the sharing, for example, a single-producer, multiple-consumer situation, read-broadcast would have provided more benefit for a similar cost in processor lockout.

Competitive snooping benefits only those programs in which the pattern of reference to shared data is very sequential. In this case the decline in the number of write-broadcast cycles is greater than the additional cycles introduced by invalidations and invalidation misses; the net effect is a drop in bus utilization. However, for programs characterized by fine-grain sharing, competitive snooping can degrade performance by causing a slight increase in bus utilization and total execution time. Competitive snooping works well in programs that would have incurred less coherency overhead with write-invalidate protocols (rather than write-broadcast). The reason is that it uses invalidations to terminate broadcasts to shared data.

6.7. References

[Agar88] A. Agarwal, J. Hennessy and M. Horowitz, "Cache Performance of Operation System and Multiprogramming Workloads", *ACM Transactions on Computer Systems*, 6, 4 (November 1988), 393-431.

[Alex86] C. Alexander, W. Keshlear, F. Cooper and F. Briggs, "Cache Memory Performance in a UNIX Environment", *Computer Architecture News*, 14, 3 (June 1986), 14-70.

[Egge89] S. J. Eggers and R. H. Katz, "The Effect of Sharing on the Cache and Bus Performance of Parallel Programs", *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston MA (April 1989).

[Good87] J. R. Goodman, "Cache Memory Optimization to Reduce Processor/Memory Traffic", *Journal of VLSI and Computer Systems*, 2, 1 & 2 (1987), 61-86.

[Good88] J. R. Goodman and P. J. Woest, "The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor", *Proceedings 15th Annual International Symposium on Computer Architecture*, Honolulu HA (May 1988), 422-431.

[Hill87] M. D. Hill, "Aspects of Cache Memory and Instruction Buffer Performance", Technical Report No. UCB/Computer Science Dpt. 87/381, University of California, Berkeley (November 1987).

[Karl86] A. R. Karlin, M. S. Manasse, L. Rudolph and D. D. Sleator, "Competitive Snoopy Caching", *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, Toronto, Canada (October 1986), 244-254.

[Karl88] A. R. Karlin, M. S. Manasse, L. Rudolph and D. D. Sleator, "Competitive Snoopy Caching", *Algorithmica*, 3 (1988), 79-119.

[Sega84] Z. Segall and L. Rudolph, "Dynamic Decentralized Cache Schemes for an MIMD Parallel Processor", *Proceedings of the 11th International Symposium on Computer Architecture*, 12, 3 (June 1984), 340-347.

[Smit87] A. J. Smith, "Line (Block) Size Choice for CPU Caches", *IEEE Trans. on Computers*, C-36, 9 (September 1987), 1063-1075.

7 Summary and Conclusions

This dissertation has investigated several aspects of the performance of parallel programs executing on single-bus, shared memory multiprocessors. Memory reference traces of four parallel programs were first collected, and then analyzed for their amount of write sharing and the pattern of multiprocessor accesses to the write shared data. Results indicated that the amount of write sharing, measured in numbers of memory references, was small and that there was little contention for either data or locks.

A simple model of write sharing was developed, based on the inter-processor sharing activity to write-shared data. The model was used to predict the relative coherency overhead of write-invalidate and write-broadcast protocols. Parameter values for the model, both derived from the sharing analysis and based on the costs of maintaining cache coherency under the two types of protocols (assuming an implementation similar to SPUR), were applied to the model to obtain the predictions. Architecturally detailed simulations validated the model (in this architecture-independent form) for the write-broadcast protocols. However, the model did not

accurately predict coherency overhead for write-invalidate. Successive refinements that incorporated a few architecture-dependent parameters, the most important of which was the size of the coherency unit, produced acceptable predictions.

Two sets of empirical studies were also completed. The first evaluated the cache and bus behavior of parallel programs running under write-invalidate protocols over a variety of block and cache sizes. The analysis determined the effect of coherency overhead on both cache miss ratio and bus utilization by focusing on the sharing component of these metrics. The sharing component was responsible for the parallel programs having substantially higher miss ratios and bus utilization than comparable uniprocessor programs. It increased proportionally (relative to the uniprocessor component) with both block and cache size, and for the larger cache configuration values determined both the magnitude and trend of the metrics. Miss ratios were 2.2 to 4.7 times greater with increasing cache size for most of the traces, and 15 times greater in the most extreme case. Bus utilization figures were similarly higher, with figures ranging from 30 to 70 percent of available bus cycles. Increasing block size either increased the number of invalidation misses or decreased them at a rate that was less than for uniprocessor misses. In the former case the increase was substantial enough to reverse the declining miss ratio trend that normally occurs with larger block sizes. Again, bus utilization followed suite.

The second set of studies was a cross-protocol comparison. It first provided empirical evidence of the performance loss caused by increasing the block size in write-invalidate protocols and the cache size in write-broadcast. It then measured the extent to which read broadcast improved write-invalidate performance and under what situations competitive snooping helped write-broadcast. The results indicated that read-broadcast reduced the number of invalidation misses (by 4 to 51 percent), but at a high cost in processor lockout from the cache. The surprising net effect was an increase in total execution cycles of up to 3.6 percent. Competitive snooping benefited only those programs in which the pattern of references to shared data was one of sequential sharing. Both bus utilization and total execution time dropped moderately. For

programs characterized by fine-grain sharing, competitive snooping at times degraded performance by causing a slight increase in bus utilization and total execution time.

One result was central to all studies: the importance to good cache and bus performance of the pattern of memory references to write-shared data. When the pattern is one of sequential sharing, performance, measured by a wide variety of metrics, is better than when the sharing behavior is characterized by fine-grain sharing.

The duality was evident in several ways. Modeling the pattern of memory references to shared data was the single most important factor in developing a model of coherency overhead that was accurate for both write-invalidate and write-broadcast protocols. The necessity to include a parameter that represented sharing behavior was noticed when the architecture-independent form of the model could not be validated for the write-invalidate protocols. Here the size of the coherency block in the realistic simulations differed from that in the more abstract sharing analyses. The memory access pattern to the shared data within the coherency block dominated the effects of the sharing pattern intrinsic to the program. A savings in coherency overhead occurred when the memory access pattern exhibited sequential sharing; and additional coherency cycles resulted with fine-grain sharing. (Under write-broadcast the problem was not apparent, because the size of the coherency block matched the one-word unit of access in the architecture-independent model.)

The pattern of memory references to shared data was depicted in the sharing model by a parameter for the size of the coherency unit. Simulations that produced model predictions could then track shared memory reference behavior to addresses within the coherency unit as a whole, which more accurately mimics write-invalidate protocol behavior. Incorporating this single parameter into the otherwise architecture-independent model produced results that more accurately predicted coherency overhead in write-invalidate protocols. The improvements ranged from a factor of 4.8 to 52.7, depending on the trace. They brought the model's predic-

tions of coherency overhead within 2 to 8 percent of architecturally detailed simulation values.

Sharing behavior was also pivotal in the empirical studies of the cache and bus behavior of parallel programs. For the programs analyzed, the amount of sharing overhead and therefore the coherency cost in terms of miss ratios and bus utilization, depended on the intra-block memory reference pattern for shared data. Invalidation signals in programs with sequential sharing declined with increasing block size, producing a falling miss ratio; bus utilization also fell with increasing block size, and the proportion of sharing-related bus cycles to total bus cycles was less than for programs with fine-grain sharing. Programs that exhibited fine-grain sharing had the opposite behavior. Invalidation misses rose with increasing block size; and since they comprised the majority of total misses (a larger component than for programs with sequential sharing), their miss ratios rose. Bus utilization also followed suite, and quite sharply.

The divergent memory reference behavior was also apparent with increasing cache size. Miss ratios and bus utilization of programs with sequential sharing were much more responsive to increases in cache size than those whose behavior was characterized by fine-grain sharing. For both memory reference patterns the metrics declined with cache size, but the decline was sharper for the better behaved programs. The proportion of sharing-related bus cycles to total bus cycles was greater with fine-grain than sequential sharing; their magnitude was responsible for the insensativity of bus utilization to changes in cache size for programs with intra-coherency block contention.

Sharing program behavior was important in the studies that compared write-broadcast and competitive snooping protocols. Write-broadcast protocols were designed to perform well when there was contention for shared data. Studies in this dissertation indicated that they met this goal. For all cache sizes studied (16K to 512K bytes), programs with fine-grain sharing issued fewer broadcasts than those with sequential sharing. However, whether the better broadcast performance resulted in lower bus utilization depended on the uniprocessor behavior (i.e.,

uniprocessor bus traffic) of the programs. Therefore the lower coherency costs of fine-grain sharing did not always translate into better overall performance.

The competitive snooping protocol only benefited those programs with sequential sharing patterns. For these programs the invalidation feature reduced the number of broadcasts up to 70 percent, several times more than for programs characterized by fine-grain sharing. The reduction in broadcasts resulted in lower bus utilization and total execution time. For programs characterized by fine-grain sharing, competitive snooping degraded performance, causing a slight increase in both bus utilization and total execution time.

In summary, programs that exhibited sequential sharing produced less coherency overhead in multiple studies, for all metrics and across all block and cache sizes. The results clearly demonstrate the advisability of devising techniques to deliberately allocate shared data in such a way as to produce an inter-processor memory reference pattern characterized by sequential sharing. This dissertation has suggested two alternatives for shared data reorganization. The first involves the explicit programmer specification of data that is used by different processors, and runtime support for its allocation in shared memory on cache block boundaries. The technique is a straightforward solution for reducing coherency bus traffic, but places the responsibility for optimal runtime memory usage of shared variables entirely on the programmer. The second approach relies on the automatic compiler detection and subsequent memory allocation of per-processor shared variables. The problem is difficult, because the compiler must analyze references to pointers rather than discrete variables. The technique would free the programmer from having to reorganize shared data, but at considerable software complexity. At this point no solutions have been found.

The success of the memory reorganization approach may be hindered by constraints in the semantics of the underlying algorithm. For example, the algorithm may generate inter-processor contention for data and the number of processors may be quite large. Allocating the

data to separate cache blocks would not eliminate multiple invalidations and invalidation misses. For these cases a different approach should be taken. One promising technique is to generate, again via the compiler, different coherency code, depending on the processor usage of the shared data. Invalidations would be issued when program behavior is one of sequential sharing, and broadcasts when it exhibits fine-grain sharing. Both compiler approaches are areas of future research.